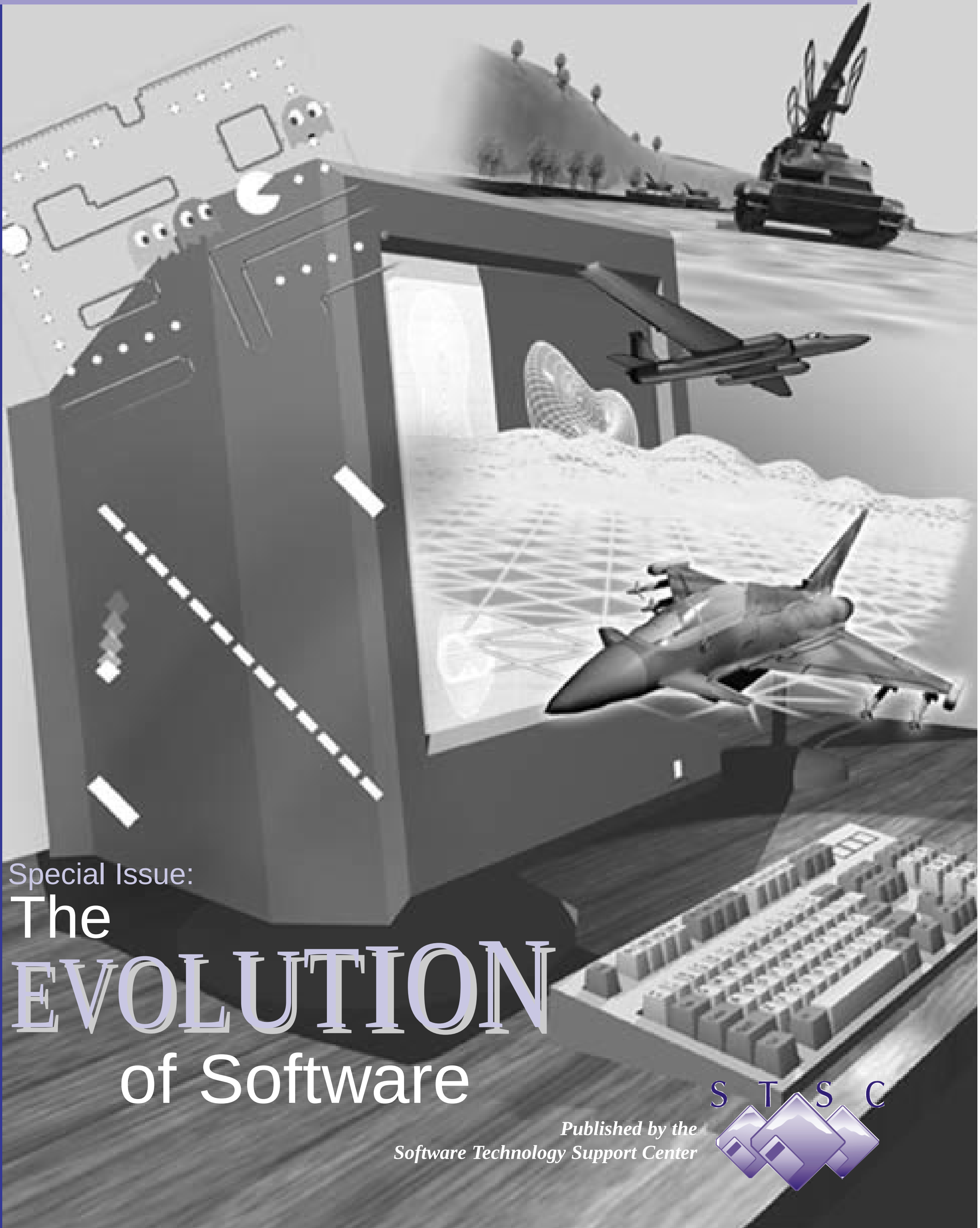
CrossTALK

The Journal of Defense Software Engineering



Sponsored by the
Embedded Computer
Resources Support
Improvement Program
(ESIP)

December 1999
Volume 12 Number 12

Special Issue:

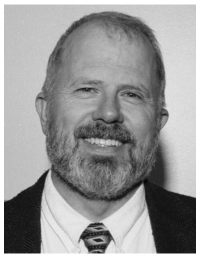
The EVOLUTION of Software

STSC

Published by the
Software Technology Support Center

Endless Possibilities

Reuel Alder
Software Technology Support Center



We started the 20th century heavily committed to the industrial revolution. We employed machines to augment our physical abilities, lifting the burden of manual labor from our shoulders. This allowed us more time to reflect upon the state of existence. Who could have predicted that our thoughts and desires, backed by generations of conditioning, would prompt the consumption of massive resources to build bigger, better, and shinier possessions. Our great-great-grandparents raised 14 children in a one-room log cabin. We raise one or two in a palace that rivals that of the kings of medieval Europe. This affluence was made possible by the industrial revolution, but it pales in comparison to the information revolution that we are immersed in now.

In the information revolution, we have learned to add artificial intelligence to the machines we create. The F-16, B2, and other systems cannot function without the aid of computers. They hold the distilled human knowledge of pilots, engineers, programmers, and others who have created software to anticipate and react to every conceivable situation. Software controls our cars, microwaves, televisions, phones, and washing machines. We fantasize of a day when computers will become self-aware like Data on the Starship Enterprise.

The information revolution is dependent upon computer systems with the capacity to store, retrieve, and analyze massive amounts of data and information. Computer systems are controlled by software created by humans using a higher order language to communicate their instructions to the computer. In this issue of *CrossTalk*, Dr. David Cook discusses the evolution of computer languages and how they have evolved to capture our ideas (page 7).

Ultimately, all computer languages must be broken down into patterns of ones and zeroes for the computer to execute them. In the computer world, a defective pattern of ones and zeros will lead to a programming error, such as the slowly degrading capability of the Patriot missiles during the Gulf War (described by Lt. Col. Scott Dufaud and Lynn Robert Carter in their article on page 14). Similarly, in the biological world, DNA stores our genetic makeup in a specific pattern that defines how our bodies function. One pattern out of place may communicate the wrong biological information, making our bodies susceptible to a number of diseases. Recent advances in genetic research, facilitated by computer software programs, have enabled the permanent repair of genetic imperfections in animals.

What does this mean to computer professionals? Are the lines between engineer, programmer, and geneticist becoming blurred? With advancements in software technology, the future of programming will conceivably extend beyond complex hardware and software to human design. The possibilities are endless.

Join us as we devote the last *CrossTalk* issue of the 20th century to the Evolution of Software and explore the past, present, and future of software engineering. ◆

See You in the Next Century

CrossTalk is pleased to bring this special issue to readers on the eve of a new century. Staff members, from left to right, are Kathy Gurchiek, *Managing Editor*; Heather King, *Associate Editor/Layout*; **back row**: Rudy Alder, *Publisher*; Heather Winward, *Associate Editor/Features*; and Tracy Stauder, *Associate Publisher*.

Level 2 Achievement

Nature abhors a vacuum, thus it comes as no surprise when an individual supplies his or her own guesses and assumptions to fill in factual gaps. Such apparently happened to the author of the Letter to the Editor in the September 1999 *CrossTalk*. As a principal participant in the Thrift Savings Plan Division's (TSPD) Software Process Improvement (SPI) effort at the USDA's National Finance Center, I would like to address the inferences drawn by that [letter writer].

First, a brief background. The TSPD develops and maintains the Thrift Savings Plan software for the Federal Retirement Thrift Investment Board under a fee-for-all-service agreement. SPI was initiated and funded by the customer in the interest of the program's million-plus participants. The customer, not the software engineers, established the immediate goal of Capability Maturity Model® Level 2, with substantial progress toward Level 3 by October 1998. The customer is also backing continued process improvement. On reflection, though, the author of the [letter] might surmise the difficulty of preventing Level 2 from becoming the primary focus of engineers whose jobs were riding on the Software Capability Evaluation (SCE) results.

The division's initial assessment was via the Capability Maturity Model-Based Appraisal for Internal Process Improvement (CBA-IPI) method in August 1995. Subsequent assessments were via the SCE. The SCEs were extremely rigorous (more so than a typical CBA-IPI), and were led by the principal author of the SCE method, Paul Byrnes. The [letter] neither stated nor implied that the first assessment occurred 10 months prior to the successful assessment. In fact, it states that, "…efforts began in earnest in November 1997 with the organization and rollout of several key processes." Prior to that date, there was a lot of motion, very little progress. The Software Engineering Process Group (SEPG) spent two years in attempts to build real management commitment, involvement, and direction. When they obtained it, with assistance from the Software Technology Support Center, the real improvements began. The real improvement journey to Level 2 took place in 10 months.

I have been involved in SPI for some five years now—successfully. I have had many opportunities to discuss elements of success and failure with many organizations conducting SPI activities. I have found job insecurity to be perhaps one of the primary motivators of engineers to adopt and practice process improvements. They and their managers tend to recognize SPI as nice in a perfect world, but are usually too deep in firefighting to spare time for fire prevention. I have found that most of the successful organizations I have talked to had to hit some type of significant low point before SPI really took hold. In a perfect world, we would all embrace SPI for the right reasons, but I have not seen that as the most common motivator for initial SPI. On the other hand, we saw and appreciated real improvements in quality derived from implementing basic project management. Are they invalidated by initially misplaced motivation?

A primary lesson learned through findings of an interim SCE, in which we failed to satisfy the verification common feature of all the Level 2 Key Process Areas, was that it would behoove an improvement organization to develop Software Quality Assurance (SQA) first. The SQA process can be developed in the absence of other software processes. As other new processes come on line, they simply become inputs to SQA. The Capability Maturity Model makes a clear distinction between quality control (inspection of the product as it rolls off the line), and quality assurance (review and inspection for fidelity to the process, procedures, and standards). Product quality remained the responsibility of the project team; SQA ensured that quality activities were planned into the projects and conducted as planned. In the latest SCE in September, the assessors had high praise for the SQA effort, finding it proactive, observed through the life cycle, and respected by practitioners to the degree that SQA personnel were sought out for consulting throughout the project.

A second lesson learned via interim SCEs: it would be difficult to "game" an assessment. To know enough to present a consistent picture of the software processes and practices, and to be able to back assertions of institutionalization with artifacts, managers and practitioners have to be practicing the improvements. Each member of the software organization has to know his or her roles on the project as well as being familiar with the roles of others. To go to the lengths it would take to achieve that result, and not implement and institutionalize the practices, would be monumentally absurd.

—Linda Giffen,
SEPG Leader,
USDA National Finance Center

Chuck Stenzrude,
SQA Leader,
USDA National Finance Center

Capability Maturity Model and CMM are registered in the U.S. Patent and Trademark Office.

On the cover: Cover artist Anthony Peters has worked as a professional artist since he graduated from the University of Utah in 1995. His work has appeared in movies, books, web sites, multimedia CDs, and television commercials. His most recent ventures include 3-D graphics and computer animation. He is a full-time artist for L-3 Communications. Anthony lives in Layton, Utah.

Up Close with General Lester L. Lyles

Kathy Gurchiek
CrossTalk *Managing Editor*

CrossTalk *recently caught up with Gen. Lester L. Lyles, Vice Chief of Staff, Headquarters, for the Air Force in Washington D.C. The general talked about the role software has played in the Air Force, and his plans to leverage off of software technology advancements in the new millennium.*

Lyles entered the military in 1968 as a distinguished graduate of the Air Force Reserve Officer Training Corps program. He received his master of science degree in mechanical and nuclear engineering in 1969 from the Air Force Institute of Technology Program, New Mexico State University. He also is a graduate of the Defense Systems Management College, the Armed Forces Staff College, the National War College, and the National and International Security Management Course at Harvard.

His various assignments include program element monitor of the Short-Range Attack Missile in 1974, and special assistant and aide-de-camp to the commander of Air Force Systems Command in 1978. In 1981 he was assigned to Wright-Patterson AFB in Ohio as Avionics Division Chief in the F-16 Systems Program Office. He served as director of tactical aircraft systems and director of the Medium-Launch Vehicles Program and space-launch systems office. In 1992 he became vice commander of Ogden Air Logistics Center, Hill AFB in Utah, where he served until 1994. He commanded the Space and Missile Systems Center at Los Angeles AFB until 1996 and became the director of the Ballistic Missile Defense Organization in 1996. He was promoted to his current position in July.

CrossTalk: What kind of advancements have you seen in the Air Force since you entered it in 1968?

Lyles: Technology has rampantly exploded in almost every mission area for the United States Air Force, and for the Department of Defense in general. Software and software-intensive systems and computer-based systems have been part of the backbone for that growth. Everything we do today is dependent on microprocessors. To some extent software has become the glue that holds everything together. We are totally dependent on good, mature, viable, testable software and I do not think people really appreciate how much software has become the backbone of everything that we are doing.

One of the difficulties we have in software management is we tend to treat it as somewhat ethereal because it is something you do not see. We have some basic management techniques we use for hardware systems, and we forget that sometimes when it comes to software, people do not know how to treat it, they do not understand it. In terms of managing it, we almost have to treat it the same as hardware systems. The fact that it is not visible is sort of a blessing but it also is a curse.

CrossTalk: One of your goals is to enhance the Air and Space integration in the Air Force. How large a role will software play?

Lyles: Tremendous. When we talk Air and Space or aerospace, and the integration of air and space missions and systems and capabilities, the space realm is the one that probably is more dependent on good software and processing than anything else we do. Since we do not normally have the opportunity—as with space shuttle programs or space shuttle satellites—to operate a man in space, we are dependent on automated systems. Those systems have decision processes or decision processors, and various other computing processors, all of which have the basic software language that makes them operate. As we grow more and more in our space capabilities, dependence on software is going to grow exponentially.

CrossTalk: If you had a crystal ball and looked ahead at software and your goals, is there anything specific that you see?

The biggest challenge of some of the contractors that we have dealt with was software development and keeping good software programmers.

Lyles: Two things. One: software development. Two: Use of processing systems.

All the different mission areas are going to get more complex in the future. We are going to be dependent more on automated systems. Unmanned aerial vehicles, unmanned aerial combat vehicles, seem to be growing significantly in today's age. We are going to be depending even more so on them and without having a man in the loop, rapid processing systems and capabilities are going to be almost mandatory to actually make those things perform properly. Software will be extremely important to make sure we can get them to do what we want them to do on a reliable basis.

I wish that somehow we could break the code in terms of how to manage software development. Before I took over as a

Vice-Chief, I led the Department of Defense's ballistic missile defense system program—often referred to as the old Star Wars program. I cannot tell you how many of our programs were hindered, delayed, over-cost, and way over schedule primarily because of poor software management, poor software development. Contractors always misestimated what it took to develop software.

I have no idea—with all the advances we have made in the last two years—why we cannot figure out exactly how many lines of code you need to do basic functions, develop that capability, get it tested, and then field it as rapidly as we possibly can. I daresay that if you were to do a trace of major problems behind most of our systems that we develop today, software probably will come out—if not the top—very near the top of the factors of why things cannot get done quickly.

The Secretary of the Army [Kenneth C. Royall], went to the entertainment industry to ask it to help develop simulators—training devices—for the things that we are doing or that he is doing in the Army. The things that they are doing for games have a lot better fidelity than some of the things we use to train our troops.

CrossTalk: Is it because programs are underestimated in terms of the budget, and in terms of the time needed to complete the project successfully?

Lyles: It is a poor estimation of how many lines of code and how complex the lines of code need to be. And the amount of time to develop those lines of code, get them into a system, get them tested, get the bugs out, and be able to operate it.

Part of that problem for the United States Air Force, and I daresay the entire Department of Defense, is that we are losing software developers rapidly. The commercial industries, entertainment industry, all the other industries out there [need software developers] because more and more depend on process capabilities also.

The biggest challenge of some of the contractors that we have dealt with in the ballistic missile defense program, as an example, was not the hardware development. It was software development and keeping good software programmers. Some companies like Lockheed Martin in Sunnyvale, Calif.—in the heart of the Silicone Valley—had a difficult time keeping good software developers, code developers, and anybody who could test [software]. They were always being lured away by some new start-up company or Internet company.

CrossTalk: So how do you keep them?

Lyles: Probably the best solution for us in the Department of Defense is to learn how to take advantage of commercial software a lot more. We all talk about reuse of off-the-shelf software, reuse of software that has already been developed for other purposes. We talk about it, we give it lip service. I am not sure how much we actually take advantage of it. I know we need to do a lot better job in that regard because we just will not be able to attract the kind of people we need to take on that very important function.

CrossTalk: What plans do you have to leverage off software technology advancements as we enter the new millennium?

Lyles: We need to do a better job of leveraging commercial software development — particularly code developers [and] software engineers. We are going to have a difficult time motivating people just to come on board to do military-related or defense-related business. We have to do a better job of leveraging the technologies, leveraging the software capabilities for commercial entities, and figure out how we can reuse it for military defense applications. Or modify it slightly so it is not a major redevelopment and recoding [effort] and still get the job accomplished.

CrossTalk: With today's trend of outsourcing software acquisition, development, and maintenance functions to contractors, what new roles do you see the 21st century United States Air Force software engineer performing?

Lyles: Management—how to manage in an environment like that. We are already beginning to experience it. I will use the example of our old Star Wars program. We found that we could not attract people to work on defense-related software development programs with a major company, and it had to subcontract its software development to smaller companies, smaller entities in the Silicon Valley area. The company could not afford to hire on and could not retain the software developers in its own company. It had to subcontract that function to small software development houses. That is a whole management entity that is new to all of us—new to industry and new to the Department of Defense. We know how to do it with hardware; it is not unusual to have a major company like Lockheed Martin or Pratt Whitney subcontract to a small vendor to do a part of the development of a hardware piece and then integrate that into a larger whole. We need to figure out how to do that with software also, because we will not be able to depend on a "Lockheed" or a "Boeing" to develop everything internally. They are going to be subcontracting code developments to smaller companies. We need to figure out how to manage that, how to integrate that, to give us the capability we need.

CrossTalk: How do you do something like that?

Lyles: I go back to my earlier comment that we tend to treat software as being special—[keeping it] almost at arm's length—in part because people do not understand it. I think we need to take a lesson learned from how we do hardware subcontracting and how we manage that. We need to take the best of those capabilities and apply them to software development.

CrossTalk: How are you planning to use software to retrain the troops?

Lyles: Software is a major part of all of the simulation modeling and simulation systems that we are trying to expand to help us in our education and training activities. As our budgets go down or stabilize, we are finding we are not able to do as many things as we used to, like flying itself. Simulators are becoming more and more important to us in everything we are involved in. I think we are going to see an explosion in the modeling and simulation industry. We have talked about it a lot in the Department of Defense over the last couple of years. We supposedly have some joint modeling and simulation activities under way but they have not really taken hold yet. We will be dependent more and more on modeling and simulation.

The Secretary of the Army [Kenneth C. Royall], went to the entertainment industry to ask it to help develop simulators —training devices—for the things that we are doing or that he is doing in the Army. Going to the [Steven] Spielbergs, going to [George] Lucas, going to those companies to try to take advantage of that. The things that they are doing for games have a lot better fidelity than some of the things we use to train our troops. I love the idea that we are trying to tie-in to the entertainment industry.

The fear is that some of these entertainment companies may not want to help work defense-related programs, so the jury is still out as to whether or not it is going to be successful. But if you are focusing on training and simulation, maybe you would not scare people away who do not want to be involved in war-related activities.

Software is the language that makes all of that happen. It is really a major part—sort of the unsung hero—of all the things we do and all the missions we try to accomplish.

One of the things we are not able to do, or cannot do as much, is fly complex missions [due to] budget, space training areas, and time. If we have a complex mission today that involves several different airplanes of several different types, it is very hard to get all those planes together to train a mission. We are now trying to figure out ways of doing sort of "distributive simulation" and have them all tied-in with very complex software and computer systems so that they, in real-time, can train with each other even though they are all over the country.

We are doing more of that and you will probably hear a lot more of that in the future. It might even become the way we do most of our training. ◆

What was the best and/or worst software technology innovation of the 20th century?

In preparation for the new millennium, CrossTalk posed this question to its readers. Here are their responses.

Best

- *the concept of context-free grammars for compilers*
- *WYSIWYG [What You See Is What You Get]*
- *binary computer language and the microchip*
- *databases*
- *MS Word*
- *C++*
- *CD player in personal computers*
- *the computer*
- *Internet*
- *MS Windows*
- *television*
- *e-mail*
- *MULTICS (Multiplexed Information and Computing Service. It is a mainframe timesharing operating system begun in 1965 and still in use today)*
- *computers for scuba diving*
- *machine code*

W orst

- *Internet*
- *C++*
- *e-mail*
- *World Wide Web*
- *the computer*
- *MULTICS (Multiplexed Information and Computing Service. It is a mainframe timesharing operating system begun in 1965 and still in use today)*
- *MS Word*
- *MS Windows*
- *television*
- *ENABLE software program*
- *databases*

Evolution of Programming Languages and Why a Language is Not Enough to Solve Our Problems

David Cook
Software Technology Support Center

Over the last 50 years, programming languages have evolved from binary machine code to powerful tools that create complex abstractions. It is important to understand why the languages have evolved, and what capabilities the newer languages give us. This article reflects on the capabilities and features of each generation of programming languages. It evaluates each language in terms of the software engineering concept of abstraction from both a data and machine viewpoint, and shows how more powerful languages can build higher-quality systems. The concept of the topology of a language also is discussed, along with the inherent limitations of a programming language.

"As long as there were no machines, programming was no problem at all; when we had a few weak computers, programming became a mild problem and now that we have gigantic computers, programming has become an equally gigantic problem. In this sense the electronic industry has not solved a single problem, it has only created them — it has created the problem of using its product" [1].

E.W. Dijkstra spoke these prophetic words almost 28 years ago in his Turing Award lecture. At that time, the 'gigantic computers' he spoke of probably had between 64 and 128 kilobytes of real memory, and at best a few megabytes of direct access storage devices. If he thought that the problem was gigantic then...

Generations

Most books and articles on the history of programming languages tend to discuss languages in terms of generations. This is a useful arrangement for classifying languages by age. I agree that whenever a few of we 'more mature' software engineers get together, we cannot ever seem to agree on what constitutes the generations of computer languages. We know that Formula Translation (FORTRAN) was probably a first-generation language. Does that make FORTRAN 77 and WatFor second-generation languages? Is the newest FORTRAN (FORTRAN 90) third or fourth generation? How about Common Business-Oriented Language (COBOL)? It has been around since 1959, and yet COBOL 2000 will be an object-oriented (OO) COBOL. Does this make it fourth generation or is it still first generation? I have the feeling that an OO-COBOL is either the e^{th} or π^{th} generation, or perhaps some other transcendental number.

To prevent numerous readers from e-mailing me and telling me that I do not understand the basic facts on the history of programming languages, I will pass on this phase of the discussion. There are numerous articles and books on the generations of programming languages. For immediate gratification, reference *Byte Magazine* online for an article on "A Brief History of Programming Languages" [2].

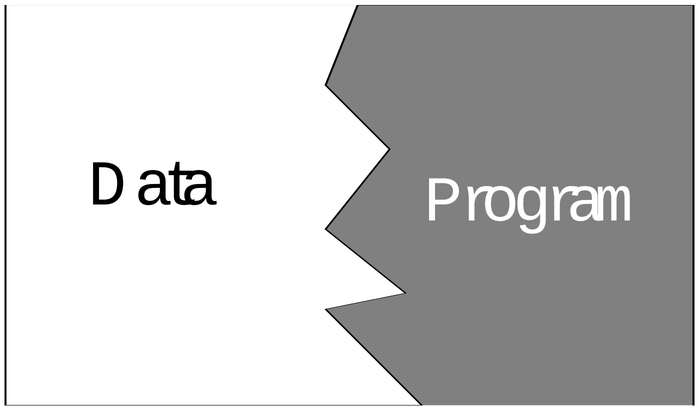
Data Abstraction and Topology

One of the keys to successful programming is the concept of abstraction. Abstraction is the key to building complex software systems. A good definition of abstraction comes from [3], and can be summed up as concentrating on relevant aspects of the problem and ignoring those that are not currently important.

The psychological notion of abstraction permits one to concentrate on a problem at some level of generalization without regard to irrelevant low-level details; use of abstraction also permits one to work with concepts and terms that are familiar in the problem environment without having to transform them to an unfamiliar structure.

As the size of our problems grow, the need for abstraction dramatically increases. In simple systems, characteristic of languages used in the 1950s and '60s, a single programmer could understand the entire problem, and therefore manipulate all program and data structures. Programmers today are unable to understand all of the programs and data — it is just too large. Abstraction is required to allow the programmer to grasp necessary concepts. To understand how abstraction works, it is helpful to show the topology or mapping of a language to the data structures and program modules that the language provides. Once we see the topology of early languages, we can better understand the problems and solutions. The clearest comparison between languages and their topology that I have discovered can be found in [4]. The material on topologies of languages comes from his book.

Figure 1. *The topology of assembly and machine languages.*



Topology of Assembly and Machine Languages

Early languages (pre-FORTRAN and pre-COBOL) had little distinction between programs and data (see Figure 1). The data and program co-existed. Because the data and program often were ill-defined, the boundary was irregular. It often was hard to distinguish between data and code. If a true hacker needed to use the number 62, and he or she knew that the Instruction BGD (Bump and Grind Drum) was coded as a hex 4E — which is equal to decimal 62 — the hacker would reference the BGD instruction elsewhere in the program to refer to decimal 62. Sounds confusing? It was. To understand how difficult programming and debugging could be with this topology, refer to "The Story of Mel, a Real Programmer" in [5] under Appendix A, "Hacker Folklore." Without any abstractions, the programs were complex, difficult to debug, and almost impossible to modify. Some structure was needed.

Topology of Early Languages

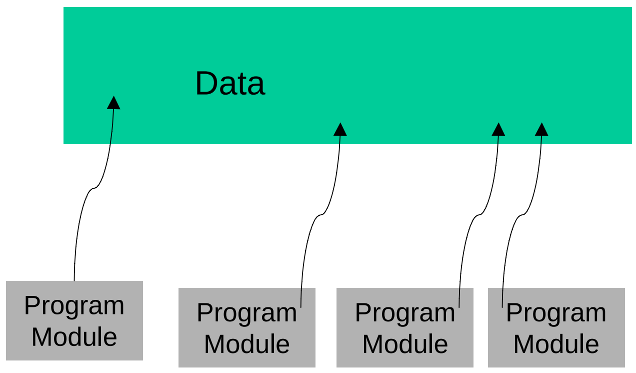
The first programming languages widely used, such as the early version of FORTRAN and COBOL, had a clear separation between the data and the program. These languages had a global data structure, but permitted modularization of program structure. Typically, there was only single-level modularization of the program (see Figure 2). While this separation of data and program was a good thing, all program segments were at a single level, and typically referenced each other in very complex ways. In software engineering terms, the cohesion was typically very low, and the coupling was quite high [6]. In other words, modules tended to perform many tasks, and there was a lot of dependence on the workings of other modules. In addition, each module had unlimited access to all data because the data was global to all modules. Global data is bad — it makes maintenance extremely difficult, since it is hard to determine which module is 'trashing' the data.

To build larger and larger systems, which were more and more complex, improvements were needed to make large-scale development easier.

Topology of Later-Generation Languages

Most of us who learned to program in the 1970s and '80s learned to program with languages such as PL/1, Pascal, ALGOL, later versions of FORTRAN, or C. These provided hidden modules within larger modules, which led to an increase in cohesion; it was easier to put logically related routines together and a decrease in coupling. This resulted in a marked increase in maintainability and robustness. See Figure 3 for an example of the topology of these languages. However, the same problems that existed for earlier languages remained, because there were no improvements on the data's topology. Because the data was unsegmented and unprotected — all modules that had access had full access to read and modify the data — problems still existed. What was needed was a way to enforce limitations on the data so that effective abstractions could be created and enforced.

Figure 2. *The topology of early languages ('50s and '60s).*



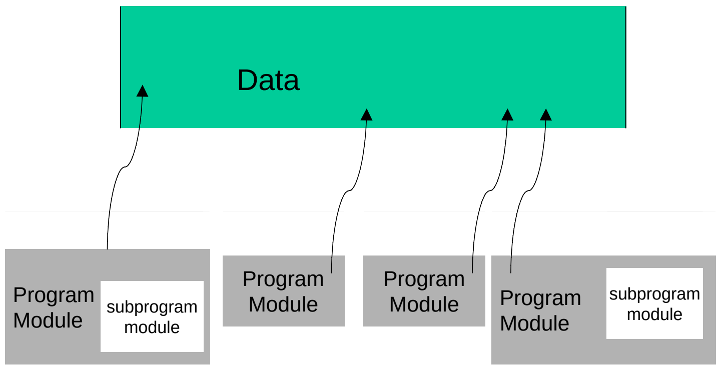


Figure 3. *The topology of later languages ('70s and '80s).*

Topology of Modern Languages

Most of us who learned to program in the 1990s were exposed to languages such as Ada, Ada95, or C++. Languages such as these permit abstractions in both the data and the program units, which in turn permit us to not only create abstractions, but also enforce the abstractions. In Ada and Ada 95, the concept of private types permit sharing of data that has limited, and compiler-enforced, restrictions on the use of the data. In C++ and Java, classes can define method access, such as public and private (refer to Figure 4). These modifications to the way we access data allow us to create powerful abstractions, and then control the way the data is accessed. We can even control who can access the data, and limit access to certain program modules.

What did the increasing complexity of topology gain us? To ask it another way, how has the evolution of program and data abstractions helped us do our job? The answer: it has allowed us to "abstract away" more and more low-level knowledge about the solution domain, and concentrate more on the solution domain.

Languages Shape How We Think, But so does Experience

I have always thought that there are only two types of problems in this world — complex problems and simple problems. Those problems that I can understand are simple; the rest are complex. By the same analogy, there are two types of solutions — simple and complex. Simple solutions are those I could have come up with, given time; complex solutions are those that are beyond me, and require me to learn or expand my knowledge. There often exists a disconnect between these types of solutions and

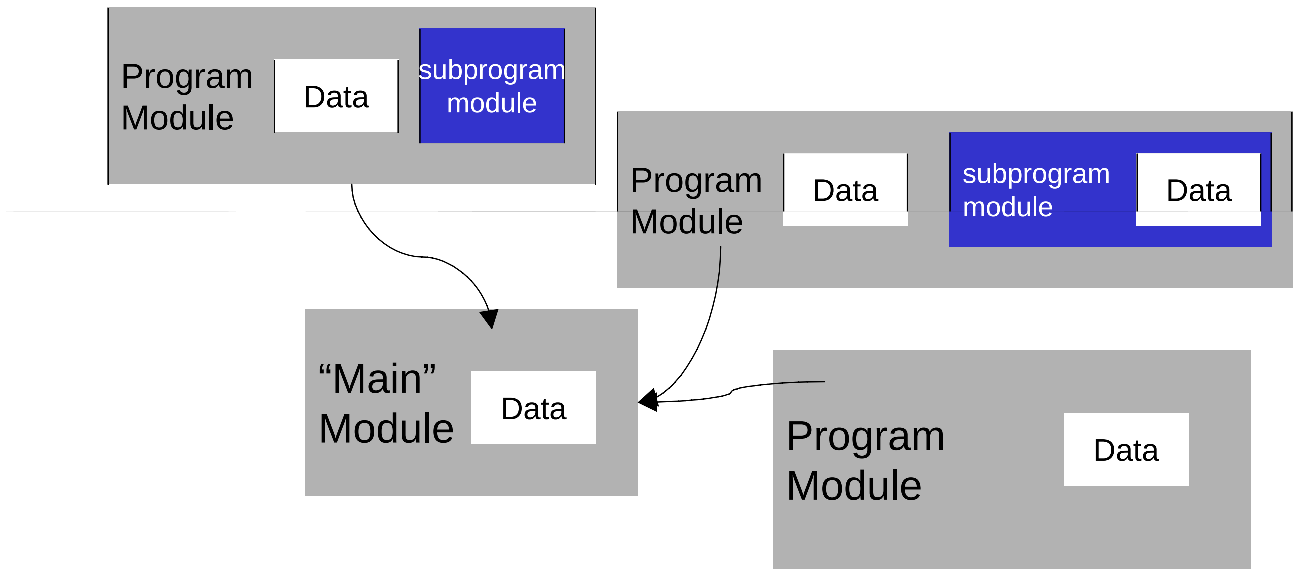


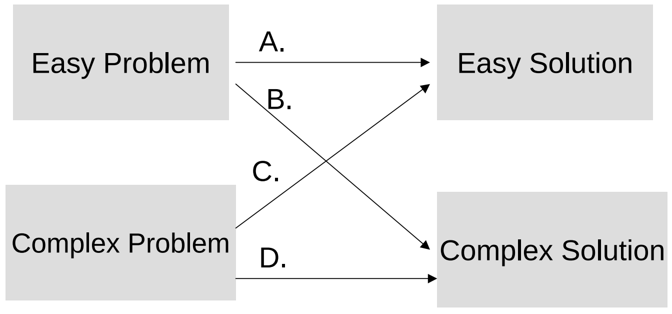
Figure 4. *The topology of modern languages.*

problems. Referring to Figure 5, we begin to see the problem.

Most of us learn to program while in college or (for Department of Defense types) at a technical school. We are first given simple problems. What we want is Figure 5, Line A (Easy Problem - Easy Solution). What we get is Line B (Easy Problem - Complex Solution). Why? Because we simply do not know enough yet to be able to see the solution in terms of the problem. I remember giving a simple sort assignment to a beginning programming class in the mid-1980s. I expected a simple bubble sort of a 10-element array. Imagine my surprise when one student turned in 40-plus pages of code. It worked, but it was so much more complex than necessary. Once the student understood how simple the problem was, and my expectations, his solution dropped to one page. Moral of the story: It takes understanding to be able to see the simple solution when you are struggling with a language at the same time.

After college or tech school ends we are thrust into the real world. There we are faced with complex problems. We expect to achieve Figure 5, Line D (Complex Problem - Complex Solution). Unfortunately, nothing we learned prepared us for extremely complex problems. We instead produce Line C (Complex Problem - Simple Solution). Our lack of understanding the complexities of the complex problem force us to only code a solution to the parts of the problem that we understand.

Figure 5. *The real world.*



What does this have to do with programming languages? It is simple. "The tools we use have a profound (and devious) influence on our thinking habits, and, therefore, on our thinking abilities" [7]. Imagine that you have the task of writing an operating system for a real-time embedded processor that will eventually run the heads-up display for an aircraft. Your only programming language experience has been with FORTRAN and COBOL. Can you even comprehend the concept of parallel processing? How can your mind formulate a design when critical concepts necessary to the solution of the problem are foreign to you? On the other hand, assume that you understand Ada. Tasking is an integral part of all Ada implementations, so the concept of parallel processing is part of the "necessary tool set" that your brain has been trained to think with.

In light of the concept of "we program based on what concepts the programming language provides," let us re-examine how programming languages have evolved.

Ignore the Solution – Concentrate on the Problem! Languages as Abstractions

In the beginning, we coded in machine code. Given a problem, typically the implementation of the solution was harder than the problem. An in-depth understanding of the machine was necessary for even the simplest problem. Most of the effort was expended in learning how to program. The hardest problem was coding a program, not meeting the users' needs. (See Figure 6). The block labeled "machine code" still leaves the programmer far from the users' needs. The inherent barrier to understanding and solving the user problem is vast. The programming language used (machine code) is so hard to use that most of the programmer's efforts is spent understanding the language, not the problem. The language used does nothing to help the programmer break through the wall of understanding. Because machine language is so far from the users' needs, changes to the user needs will take a long time and great effort to implement.

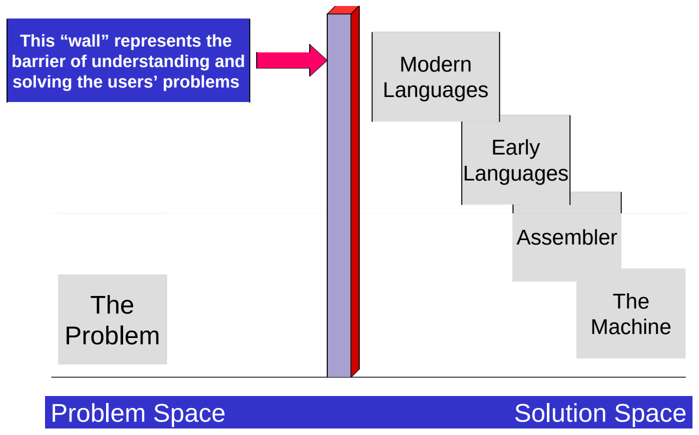


Figure 6. *Languages.*

Hiding the Machine – Assembly Language

The first major programming improvement was the abstraction of the machine. Instead of learning what bit accomplished what tasks, I could instead learn mnemonics that stood for bits. For example "LDA" for "LoaD the A register" was a lot easier to memorize than hex 47. Learning that `PRINTIT "test"` actually called a channel program that transferred a string to the printer was much easier than learning to program an input/output channel. You were still tied to the machine, but you could spend less time learning to program, and a bit more understanding the problem. I am getting closer to the problem by spending less time understanding how to understand the solution. Figure 6 shows that assembly language "abstracted away" the need to understand the hardware intimately.

Hiding all Hardware – Early Programming Languages

The next obvious step was to totally hide the hardware. In a sense, the first generation of programming languages, such as COBOL and FORTRAN, made the programmer hardware-independent. For the most part, early programming language brought you one step closer to the solution by hiding any direct reference to a real machine. You were one step closer to understanding the problem — you no longer had to concentrate on the machine to understand the problem. More time could be spent understanding the problem, and almost no time spent understanding the hardware. More of our knowledge can be put to work solving user problems, rather than learning and remembering arcane machine operations.

Getting Closer to the Problem – Using Real-World Data and Programming Structures

Modern languages, such as C++ and Ada, not only allow abstractions, but permit the enforced implementation of restrictions on abstractions. Most modern languages are object-oriented, which allows me to model the real world using my language. In addition, I can limit access to model real-world restrictions on data. The key is that I used the term "real world." For the first time, I am concerned with modeling my solution in terms of the problem. I want my solution to be problem-oriented, so that the solution reflects the real world in terms of data structures and access to the data. I also can directly implement and model real-world objects using classes (in C++ or Java) or packages (using Ada). Figure 6 shows that I am concerned with analyzing the problem in a manner that my coding will directly reflect. For the first time I can realistically try and cross the "wall" to the user, and try and capture the real-world problem in terms that I can directly reflect in my code.
Limitations on access and "hidden" information can be captured in my design and code. In addition, because of these language features, I can concentrate more on what a problem is rather than how the problem will be solved. The concepts of reuse in C++ and Ada and other similar languages permit me to create a standard library of routines common to a problem space, and reuse these routines when convenient.

Understanding the Real Problem

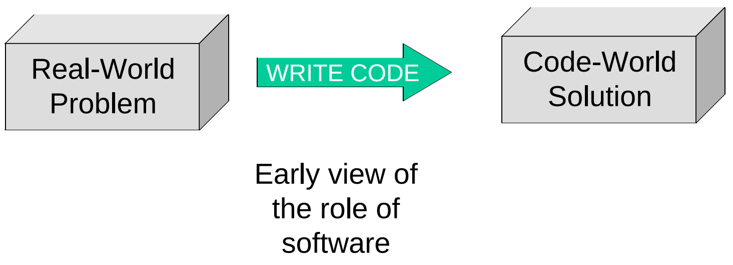
Prior to modern languages, we often viewed software as the actual solution to a problem (see Figure 7). Software is not the solution. In fact, as many seasoned practitioners already know — software is often the problem!

We have reached the crux of the matter. The real problem in programming is (and always has been) understanding the problem, and trying to implement it directly in code. Early attempts (using machine and assembly language) added extreme complexity by requiring knowledge of the implementation hardware. Using current languages, however, I can implement my solution by concentrating more on the problem. Real-world entities can easily be modeled and abstracted using objects. The biggest difficulty in solving a problem is studying the problem domain and making discoveries about it [8]. Modern languages give us the luxury of being able to spend more time on the problem, and less on the solution. I am not afforded the luxury of being able to design my solution in terms of the problem, and then implement it using the same design.

As a further note, most practitioners of software engineering understand that the code is not the final solution. In modern, large-scale systems, our code is usually a small piece of the overall system. Because of the large-scale problems we are solving, we also are faced with the task of integrating the "code solution" into a larger overall system. This complicates our task, as we now create only pieces of the solution, not the entire solution. This system integration forces us to acknowledge that our "code-world solutions" must directly map back into the real world and interact with many other entities (Figure 8).

Dijkstra was right. We have created more difficult problems for ourselves because our more powerful languages permit us to

Figure 7. *An incorrect view of software.*



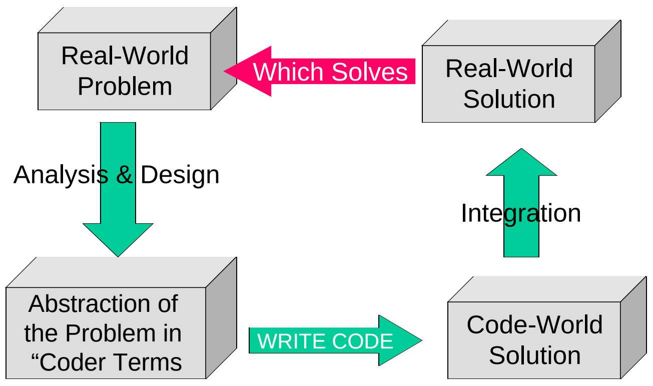


Figure 8. *Correct view of software.*

solve more complex problems.

Why We Still have a Long Way to Go – the Problem is not in the Language

Note that in Figure 6, the gap between the problem and solution is not closed. We have not reached the point where we can solve problems in a reliable manner.

First, notice that there has been a vertical bar separating the problem space from the solution space in Figure 6. This bar represents, to me, the limitations of any programming language. To be blunt, a language is just a language. The vertical bar represents the "understanding gap." To bridge the gap, such topics as requirements analysis, design, verification, and validation must be considered [9]. The languages we have are sufficient for the problems we are currently solving. If your organization is considering switching from one programming language to another, I have one word of advice: stop. I will place a large wager that your problems are not language-based, but are based on improper processes and requirements engineering.

Do You Really Want To Solve the Problem?

Here is the complex solution to a complex problem — bridge the gap with quality tools and techniques, not language (see Figure 9). If you are interested in bridging the gap between modern languages and user needs, stop worrying about languages and concentrate on understanding requirements by focusing on analysis, tools, techniques, and processes. Consider object-oriented methods. Acquire and train users in modern tools and methods. Require conformance to a process. Use the Software Engineering Institute's Capability Maturity Model® (CMM®) to implement a process. Train your programmers to be productive by using the Personal Software Process and the Team Software Process (PSP and TSP). Use metrics. Use meaningful metrics. Use common-sense metrics. Review the metrics, and change your processes accordingly.

To put it another way, we already have the tools and knowledge to code. If you are having coding problems, it is probably because you do not follow common-sense standards. If you want to have a simple checklist, I highly recommend *201 Principles of Software Development* by Alan Davis [10]. Principles 87-106 give guidance on coding. Some are simplistic (Principle 105: Format your code, and Principle 88: Avoid Global Variables). They are simple, yet almost every consulting job I have undertaken over the last two years has managed to profoundly break these two simple principles. Does your organization have coding standards? Are they enforced? When is the last time you looked at your code? Do you ever look at the code your contractors provide?

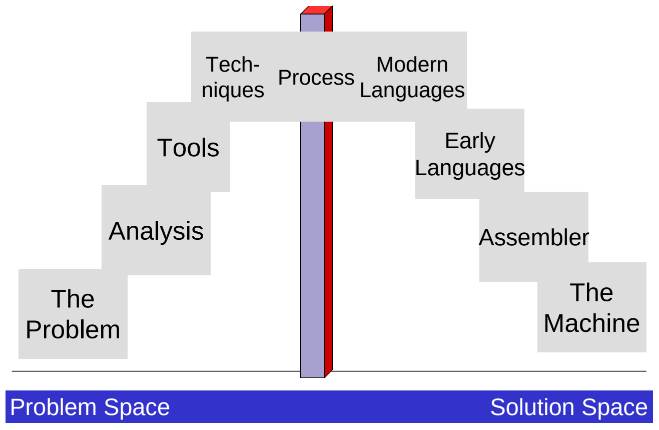
Improving Your Situation

My opinion is that current language is sufficient for our needs. Object-oriented languages are good and here to stay. C++ and Ada95 are two languages heavily used in the Department of Defense, and both rely heavily on object-oriented features to produce robust code that models the real world.

Despite the earlier slam on COBOL, object-oriented COBOL will bring object-oriented processes into a predominately non-OO-oriented data processing arena. Sure, many software engineers make fun of COBOL and COBOL programmers, but the bottom line is that the COBOL language has changed drastically over the last 40 years to allow it to meet increasingly complex customer needs. Adding object-oriented features will increase its usefulness and allow developers to create more robust code. In addition, it will ensure that the millions of lines of legacy COBOL code will continue to work. There is the potential that the multitude of COBOL programmers will be able to incorporate the advantages of object-oriented design and coding without having to retrain and learn another language [11].

What we will see in the future is less reliance on the language, and more on the modeling tools, such as the Unified Modeling Language (UML). The output of the modeling tool will produce much of our code for us; at the very least, it will produce architectural and design models and the structure of our code. This will produce a design (and possibly code) that can be validated by the customer prior to complete implementation and testing. As everyday problems that we solve become larger, we have less and less time to "redo" the code. The days of saying, "we will just write a Beta version and the customer can then tell us what they really want," are past. Organizations that fail to get complete and correct customer requirements prior to writing the code will go out of business.

Figure 9. *What comes next?*



Why? Because it takes too long, and costs too much, to write code two or more times. Organizations that have a commitment to verification and validation prior to producing code will prosper — others will fail. One of the biggest problems today is communicating with domain experts (users) [12]. Use of a tool to facilitate this will help you stay in business. If your current customers are not involved in the validation of your requirements (and probably your design), you are headed for trouble. If you cannot get your customers to understand the methodology, consider using things such as use-cases [13] and scenarios [14] and to allow customer input and validation.

Finally, remember that a programming language is a small part of the overall software development effort, which in turn is just part of the systems development. We are not really having a problem coding a solution — we are having a problem understanding what solution to code. If you want a quality product, you cannot start with the coding. Concentrate on having a quality process, such as the (CMM) [15]. Focus on good requirements engineering and a quality process. Use and follow a good life cycle model, such as spiral or UML. If you focus on requirements and verification and validation, the coding will take care of itself.

About the Author

David Cook is a principal member of the technical staff, Charles Stark Draper Laboratory, under contract to the Software Technology Support Center. He has more than 25 years experience in software development and has lectured and published articles on software engineering, requirements engineering, Ada, and simulation. He has been an associate professor of computer science at the U.S. Air Force Academy, deputy department head of the software engineering department at the Air Force Institute of Technology, and chairman of the Ada Software Engineering Education and Training Team. He has a doctorate degree in computer science from Texas A&M University and is an SEI-authorized PSP instructor.

Software Technology Support Center
7278 Fourth Street
Hill AFB, Utah 84056
Voice: 801-775-3055
Fax: 801-777-8069
E-mail: david.cook@hill.af.mil

References

1. Edsger. W. Dijkstra. "The Humble Programmer" (Turing Award Lecture), Communications of the ACM, Vol 15, No. 10 (October 1972).
2. *Byte Magazine* 25th Anniversary issue. Located online at http://www.byte.com/art/9509/sec7/sec7.htm. Refer to http://www.byte.com/art/9509/sec7/art19.htm for the article entitled "A Brief History of Programming Languages."
3. Wasserman, A. "Information System Design Methodology" *Software Design Techniques*, P. Freeman and A. Wasserman (eds). 4th Edition, IEEE Computer Society Press, 1983.
4. Booch, Grady. *Software Engineering with Ada*, 2nd edition. Benjamin Cummings, 1987.
5. Raymond, Eric. *The New Hacker's Dictionary* MIT Press, 1983.
6. Roger Pressman. *Software Engineering: A Practitioner's Approach*, 4th edition. McGraw Hill, 1997.
7. Dijkstra, Edsger. W. *Selected Writings on Computing: A Personal Perspective* Springer-Verlag, 1982.
8. Coad, Peter and Edward Yourdon. *Object-Oriented Analysis, 2nd Edition*. Prentice Hall, 1991.
9. Van Buren, Jim and David Cook. "Experiences in the Adoption of Requirements Engineering Technologies," *CrossTalk*, The Journal of Defense Software Engineering, December 1998, Vol 11, Number 12.
10. Davis, Alan M. *201 Principles of Software Development*, McGraw-Hill, 1995.
11. Grauer, Robert T., Carol Vasquez Villar, and Arthur R. Buss. *COBOL From Micro to Mainframe*, 3rd edition, Prentice Hall, 1998.
12. Fowler, Martin. *UML Distilled*, Addison Wesley Longman, Inc. 1997.
13. Jacobson, Ivar, Grady Booch, and James Rumbaugh. *The Unified Software Development Process*, Addison Wesley, 1999.
14. Fowler, Martin. *Analysis Patterns: Reusable Object Models* Addison-Wesley, 1997.
15. M. Paulk, et.al. "Capability Maturity Model for Software," Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pa. 1993.

Quote Marks

"640K ought to be enough for anybody."
— *Bill Gates, Microsoft*

"Our computer has never had an undetected error." — Weisert

"Men have become the tools of their tools."
— *19th century writer Henry David Thoreau*

"Man is still the most extraordinary computer of all."
— *John F. Kennedy, 1963 speech*

"But what... is it good for?"
— engineer at the Advanced Computer Systems Division of IBM, 1968, commenting on the microchip

"Computer viruses are an urban myth…"
— *Peter Norton*

"There is no reason for any individual to have a computer in his home."
— *Kenneth H. Olson, president of DEC, Convention of the World Future Society, 1977*

The EVOLUTION of *CrossTalk*

Tracy Stauder
CrossTalk *Associate Publisher*

As this issue takes a look back in time at software technology and practices, it is interesting to see how CrossTalk *has also evolved.*

Did You Know...

How long has CrossTalk *been in existence?*

11.5 years — the first *CrossTalk* was published in July of 1988.

How many subscribers did CrossTalk *originally have?*

Approximately 200

What were some of the topics addressed in the July 1988 issue?

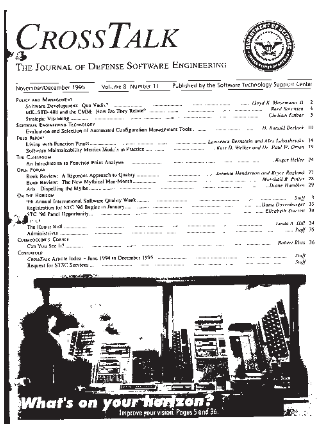
a. Artificial Intelligence
b. Recap of Ada conference for Air Force users. The conference included briefings on:
   • Standard Automated Remote to Autodin Host) — a software package which allowed a user in and out of the Autodin system without punch cards or paper.
   • Software Life Cycle Support Environment — a computer-based environment of integrated software tools, which included one to evaluate software in DOD-STD- 2167 format.
c. Software Technology Support Center's (STSC) Electronic Bulletin Board System
d. Software Engineering Institute Affiliates Program

Who was on the CrossTalk *staff in 1988?*

Bill Frost, publisher, and Susan Kelsey, editor

When did CrossTalk *become an official Department of Defense publication?*

June 1994 — Maj. Peter Vaccaro was the publisher.



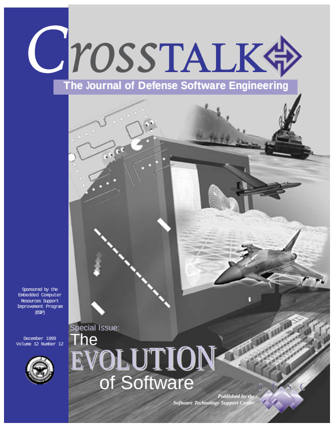
CrossTalk
The Journal of Defense Software Engineering

November/December 1995 Volume 8 Number 11 Published by the Software Technology Support Center

Policy and Management
Software Development: Quo Vadis? ... Lloyd K. Mosemann II 2
MIL-STD-498 and the CMM: How Do They Relate? ... Reed Sorensen 4
Strategic Visioning ... Christian Kimbar 5
Software Engineering Technology
Evaluation and Selection of Automated Configuration Management Tools ... H. Ronald Berlack 10
Field Report
Living with Function Points ... Lawrence Bernstein and Alex Lubashevsky 14
Software Maintainability Metrics Models in Practice ... Kurt D. Welker and Dr. Paul W. Oman 19
The Classroom
An Introduction to Function Point Analysis ... Roger Heller 24
Open Forum
Book Review: A Rigorous Approach to Quality ... Johnnie Henderson and Bryce Ragland 27
Book Review: The New Mythical Man-Month ... Marshall R. Potter 28
Ada: Dispelling the Myths ... Diane Hamblen 29
On the Horizon
9th Annual International Software Quality Week ... Staff 3
Registration for STC '96 Begins in January ... Dana Dovenbarger 33
STC '96 Panel Opportunity ... Elizabeth Starrett 30
[illegible]
The Honor Roll ... Linda A. Hill 34
Administrivia ... Staff 35
Curmudgeon's Corner
Can You See It? ... Robert Bliss 36
Centerfold
CrossTalk Article Index – June 1994 to December 1995 ... Staff
Request for STSC Services ... Staff

What's on your horizon?
Improve your vision. Pages 5 and 36.

old CrossTalk


CrossTalk
The Journal of Defense Software Engineering
Sponsored by the Embedded Computer Resources Support Improvement Program (ESIP)
December 1999 Volume 12 Number 12
Special Issue: The EVOLUTION of Software
Published by the Software Technology Support Center

new CrossTalk

When did CrossTalk *go purple?*

July 1996. The format change included adding purple color to the cover. The purple signifies the tri-service coverage of software issues affecting all United States armed forces as we all share the same software concerns and needs.

What happened to the Curmudgeon's Corner?

It never went away. In July 1997, with the help of our readers, we changed the title from Curmudgeon's Corner to BackTalk. The column makes light of the many engineering and management obstacles that are so common in the software engineering workplace. BackTalk continues to be one of our readers' favorite monthly features.

How many subscribers does CrossTalk *have today?*

More than 19,000 — although readership is hard to quantify as many of our issues are sent to libraries and organizations. We also receive comments from readers wanting to subscribe after they have repeatedly borrowed a co-worker's copy.

How many monthly hits does CrossTalk *entertain on its web site today?*

An average of 50,000 per month.

A Look Ahead

As *CrossTalk* enters the 21st century, its mission remains the same:

to encourage the engineering development of software in order to improve the reliability, maintainability, and responsiveness of our warfighting capability and to instruct, inform, and educate readers on up-to-date policy decisions and new software engineering technologies.

Upcoming themes will include risk management, education and training, cost estimation, software security, and Capability Maturity Model Integration^{SM} (CMMI). As a forum for sharing your knowledge and ideas, *CrossTalk* is counting on your continued excellent ideas and quality articles (see the Call for Articles on page 30).

Capability Maturity Model Integration and CMMI are service marks of Carnegie Mellon University.

21st Century Engineer

Lynn Robert Carter
Software Engineering Institute
Lt. Col. Scott Dufaud
Air Force Y2K Program Office

Editor's note: The following is an excerpt from a forthcoming book by Lynn Robert Carter of the Software Engineering Institute and Lt.Col. Scott Dufaud.

Foundations

It is difficult to find any single definition of engineering that generates uniform agreement, but many are similar to the following:

"*The art of the practical application of scientific and empirical knowledge to the design and production or accomplishment of various sorts of constructive projects, machines, and materials of use or value to man* [1]."

For most of history, this definition would not have been useful. It was not until the middle of the 18th century that the first civilian engineering school was created [1], and that education program was little more than a structured form of apprenticeship with very little dependence on mathematics, physics, or other related scientific fields. From this beginning to the middle of the 19th century, the addition of the scientific method, the creation of professional societies, and the growth of engineering schools led to the golden age of engineering — the middle of the 19th century to the middle of the 20th century [2]. This is not to say that many fine examples of engineering had not been created before then, but the labor was more craft than profession. Early work was the culmination of solid common sense and the wisdom of experience, while the golden age benefited from the application of scientific principles. It is also important to recall that a handful of traditional engineers would produce the drawings and guidance to potentially hundreds or thousands of skilled craftsmen on the factory floor. It was there that physical reality was created and it would be foolish to ignore the contributions of these laborers beyond just performing the work described by the engineering department.

The middle of the 20th century brought about many changes, including a rapidly growing technology base, an equally growing consumer marketplace, and the advent of computing. With this explosive growth came a shift in public opinion about engineering. The creators of the many marvels that had transformed our lives were no longer perceived as the source of solutions to society's problems [2]. While it is unfair to unload all of the blame on computing, the issues of complexity and speed of change — hallmarks of computing — along with a naïve understanding of engineering are the likely root causes. Engineering grew from a skilled craft based on apprenticeship to a profession based on rigorous engineering school education and careful development programs in a myriad of engineering departments. Little of either of these remain on the path many take in today's world of more, cheaper, faster.

The change did not happen overnight. In the early days of computing, the shift from a classical engineering discipline began. To the United States military, the computer provided solutions for a world full of lethal enemies and increasing nuclear powers. From the calculation of trajectories for big guns, to the fusion of data to detect first-strike missile launches, no mechanical or single-function machine could fit the bill. A calculating machine with modifiable functionality was precisely what was needed and the United States military took the lead. From the 1940s through the '60s, the state-of-the-art in computing was defined by the work produced by or on contract for the military. While several noted institutions of higher education provided specific technical courses in computing, the bulk of the software was developed by engineers from the classical engineering fields or by mathematicians. As the military continued to find ever more sophisticated ways to employ computing, a trend became obvious. A large portion of the total life cycle costs of systems shifted from hardware to software at an alarming rate, far faster than new software developers were being produced.

As the complexity and costs of systems shifted from hardware to software, the need disappeared for a factory of skilled craftsmen to transform the output of engineering into physical reality. A hidden downside was the loss of a critical system review by involved but independent people. It was common for factory personnel to uncover problems and resolve them before the customer saw the result. The lack of effective feedback mechanisms to engineering was one of W. Edwards Deming's complaints about North American manufacturers [3] and now the people best positioned to provide that feedback were being eliminated. Where the skillful eye and ingenuity of the factory worker had served to back up creative engineering departments, new products are assembled by compilers, linkers, and computer-driven duplicators, none of which have common sense or insight about the real need of the customer. It is true that this automation can be less costly and far faster; it assumes the output of engineering is perfect. If it is not, a critical quality control function was eliminated and few firms took the initiative to effectively place those old responsibilities onto engineering.

Prior to the shift toward software, quality was built into the product through a self-imposed process where engineering on paper and tube-bending realism had to meet and work. In addition, turning out more product was largely a manpower issue. Leaning on the manufacturing process to turn out more product only required working longer hours, working more people (maybe in shifts), or both. If the factory employed skilled workers, provided them adequate tools, and allowed them to build quality products, quality products resulted, even if the drawings from engineering were flawed. Software does not fit this paradigm. Because software quality is not a characteristic of the physical world, this aspect of the product could not be determined by visual inspection by either the producer or the consumer. Quality was no longer a partnership of the engineering department and the factory floor with its skilled craftsmen. Quality is an exercise in mental understanding of what the product is supposed to do and shaping creative possibilities into code. In the software paradigm, turning out more product also is different as the market is always looking for the next release with a set of new features. No longer does using more factory workers putting in more hours turn out more quality product. In order to do this, we must now lean on engineering. In response to the need for more software, many elected to hire more developers and compressed the time line to develop them. This has led to an even further degradation in the spectrum of quality, with fewer and fewer software developers receiving even the most basic formal training or skill development in the application of scientific methods to their work. Without the formalisms or the apprenticeships of old, where are they supposed to develop their skills?

The software crisis continues to expand at rates commensurate with the increasing complexity of our systems and technology.

Situational Assessment

We have become so used to poorly designed software and throw-away hardware that we do not even think about it. While it is common for the military to think in terms of 30- to 50-year weapon system life spans, the commercial off-the-shelf (COTS) hardware and software systems used to feed data to these weapon systems and to link them together via networks are nearly obsolete the day they are installed. In addition, the lack of long-term vendor support seldom seems to be a concern. Sadly, quality problems are not just problems in the COTS hardware and software. The destruction of the space shuttle Challenger (faulty seal design), the distortion in the Hubble images (improperly ground lens), the slowly degrading capability of the Patriot missiles during the Gulf War (software error), and the long string of Delta launch vehicle failures (cause not publicly known) are all high-profile examples of quality problems from firms from which we expected more. Rather than designing-in quality up front, the attitude of many seems to be to test quality in with huge beta-test efforts, even in the face of mathematical proof that such testing can not ensure quality.

Even more disturbing is the growing dependence of the military on technologies, such as those supporting the World Wide Web and the Windows operating system, that have been repeatedly shown to be vulnerable to even teenage hackers. What usually goes unspoken is what could be accomplished by an elite information warfare team. Since these technologies were designed for commercial applications by commercial firms addressing a commercial marketplace, many issues critical to our war-fighting military were not design requirements. As a result, our military services struggle to obtain and keep enough qualified and skilled network engineers, to keep these critical systems running, and to keep them secure. Some of these people can cost more per year than the cost of the computing systems on the network they support. Better network implementations are being pursued, but in the current climate of "more with less," these are being addressed in a piecemeal fashion, at best. Attributes like quality, standardization, and repeatability are most often found in organizations with people having strong personalities that believe in these attributes. That is to say, these organizations are personality-driven and these attributes are part of those personalities as opposed to being core attributes of the overall organization. The problem with this situation is that when the proponent leaves the organization or is forced to reorder his or her priorities, the organization's products and processes cease to possess these characteristics.

This situation results in wasted resources and increased mission risk when we can least afford it. The shift of costs from hardware to software has led to a shortage of adequate software development capacity, labeled "the software crisis," and it continues to grow at rates commensurate with the increasing complexity of our systems and technologies. While many have fought to bring the title "engineer" to the field of software development as the solution to this crisis, the pressures of the marketplace and rapid rates of technology change have resulted in an interesting split. We use the phrases "blue-collar software developer" and "gold-collar software developer" to highlight this split.

Two Paths to the Future

The trends seem clear. Consumer demand for ever more sophisticated systems at lower costs will continue to drive commercial and military development. In turn, the development community must take a similar economic approach for solvency. Survival will depend upon the ability to increase efficiency, shorten cycle time, and focus on the bottom line, without sacrificing quality. As we have indicated above, there appear to be two very different approaches to address this trend: blue-collar and gold-collar development strategies. We offer the following distinctions to differentiate the competing philosophies.

The first difference between these developers can be found in how their employers go about addressing the need for more output. Blue-collar organizations believe the solution is to hire more workers and find ways to keep the cost of each as low as possible. The hiring process is little more than verifying that the developer has the needed skills, is

The Capability Maturity Model and CMM are registered in the U.S. Patent and Trademark office to Carnegie Mellon University.

willing to work long hours, and is willing to take lower pay in exchange for stock. (It is sad to note how few engineers are able to redeem their stock for cash and recover the income they have deferred.) Gold-collar firms look for ways to better leverage their existing resources by eliminating waste, obtaining better tools, and ensuring that barriers to progress have been removed. Fast, flexible, and cost-effective solutions seldom occur on projects buried under developers and gold-collar firms always look for alternatives to the "piling-on" approach to shorten the development time. Watts Humphrey has stated that performance differences of factors greater than 10 have been seen between organizations with basically the same business, workers, and tools. This observation led to his effort to better understand these differences and to develop methods to leverage the understanding. From this came the Capability Maturity Model® (CMM®). Again and again, the Air Force and commercial firms have shown that the application of these concepts can produce dramatic results. Yet many firms still look for large numbers of people with the right design, coding, and testing skills while they ignore the nonsoftware skills required to progress up the CMM levels.

The second difference is the attitude toward process by the people in a firm. Blue-collar development firms talk about the costs and the delays associated with the overhead of process and the damaging constraints it places on creativity. Gold-collar development firms point out the dramatic improvements in productivity and quality found in a balanced focus, including process. Having reaped the benefits, they are somewhat amused by the notions that process slows development or constrains creativity. Firms that have successfully climbed CMM levels report amazing returns on their process improvement investments and point out how process reduces waste, allowing their developers to spend more time developing code likely to appear in finished systems. The results from the shuttle on-board software engineers align nicely with the results from Boeing and numerous others. Yet, blue-collar firms remain unswayed by these results.

The relationship between a firm and its customers is the third difference between these two philosophies. Blue-collar developers are kept far away from the ultimate users with buffers of managers, marketers, salespeople, and support personnel. There is a concern about loss of control and the need to keep the blue-collar developer focused on writing code. Firms obtaining full benefit from gold-collar developers see the need to have their developers fully integrated with the customer. From these firms' perspective, technology has decreased cycle time, and any nonvalue-added role in the development process can only slow things down. As technology grows in complexity, the challenge of working closely with customers grows in its importance. Blue-collar firms point to the lack of appreciation many customers have about current technology issues, the lack of clarity about product requirements, and their unreal expectations about the time and effort to accomplish the ill-defined work. These, they say, are reasons enough to maintain this distance. Gold-collar firms point to the same things and assert that this demonstrates the need to be closer. Some go even further, insisting that they must help develop their customers to ensure higher mutual satisfaction in the future. They advocate the use of the spiral model in development and in customer relationships, and trust in their gold-collar workers' use of process to keep things under control. Blue-collar firms scoff at such notions as wishful thinking and a waste of time and resources in a turbulent world. After all, they claim, we are talking about human beings, human nature, and no one can make guarantees about which firms will exist in the future, let alone who will be the market leader.

The use of scientific methods, mathematics, risk management, and lessons learned are the fourth major difference. Gold-collar workers easily separate work into well-known and novel classes of effort. Effort and work can be classified as either, "we have already done this," or "this is something new." With a solid discipline for employing process and data, they strive to leverage the work of others as well as their own experiences to build an ever-growing foundation. When results differ from expectations, they take time to understand the root cause and employ mechanisms to reduce the probability of similar difficulties in the future. Their process focus provides the basis for using statistical methods. This supports understanding common vs. special causes, allowing effectively designed improvements to address the former. Blue-collar firms point to the decrease in cycle time and the increase of complexity as proof that nothing stays the same long enough for statistical methods to be useful, even if there was enough extra resources around to waste on process and data capture.

The fifth major difference is in the perception of the skills workers need to have and the best way to organize and manage those workers. Blue-collar organizations tend toward numerous groups of specialists and organizational structures to ensure the work gets done. There are clear lines of authority and control in order to ensure no one is working on things other than what is currently required. (In many such organizations, few of those in a position of authority and control are particularly good at this. Few blue-collar firm managers are able to produce a list of projects under their control or indicate who is working on what.) Gold-collar organizations strive to populate their work force with generalists, each having one or more areas of unique skill that complements others. The goal is to have small, customer-involved, self-directed work teams. They leverage modern technologies and tools, but most importantly they leverage the other team members to do it all. They think that such tight integration of small teams leads to faster results with products that better fit the customers' needs, including making it harder for operational mistakes to occur because the common cause for such errors have been designed out of the system. (After all, gold-collar firms know how to use statistics to their benefit as well as to their customers' benefit.)

Choices to be Made

Looking at the exponential growth curves, it is clear that something will have to change. Gordon Moore, of Intel, observed that the density of storage, in bits per square inch, from integrated circuits appears to be doubling each year to 18 months. This observation has been called Moore's Law [4], and it appears to apply, in modified form, to the doubling of the number of engineers it takes to produce each new generation of Intel processors. In not too many more generations, if these trends continue, everyone in the United States will be a processor designer for Intel. This should be a clear signal that the approach of many blue-collar firms will not work. Even if they could find people to hire, many are reaching the point of diminishing returns. The easiest approach of hiring more people is not really a reasonable long-term solution. A second approach is to acquire and employ higher performance tools. At first glance, this seems more reasonable. The capability of modern software tools can be very empowering. The question that must be asked revolves around the long-term benefit of these tools. Tektronix was one of the first firms to employ high-level languages in the development of microprocessor software in embedded systems using a retargetable compiler system. The Tesla programming language was used in more than 60 products over the years, and its gamble paid off. The decision to leave Tesla for C was a difficult one, but the lack of a large enough pool of developers willing to learn Tesla forced the change. No technology will remain in first place forever. The trick is to pick technologies likely to remain useful long enough for the firm to recover its investment as well as to place itself in a position to leverage the next big technological advancement. This can be very difficult. (Ask the people who selected Beta over VHS because of the superior picture Beta delivered.)

Another aspect of a tool-based approach to the problem lies in the nature of high-performance tools. Long-term benefit from these tools requires discipline throughout the development process. Most of these tools require a heavy investment during the early phases of the life of the project in order to yield big wins at the end. While blue-collar workers may have the skills to use these tools, their firms often lack the wisdom to stay the course charted at the beginning of the project. Decisions are made with no appreciation of the impact on the project. Gold-collar firms, with their disciplined processes, are able to appreciate the impact at the end of a project to a decision being made much earlier. This does not imply they will not make bad decisions, but at least the issues surfaced and an informed choice was made. Short-term decision making is more often based on considerations that have little to do with the projects that are ultimately affected by those decisions. Political, economical, and personality-based short-term decision making is a symptom of a blue-collar mentality that says we can overcome any setback with more people and more hours. The gold-collar mentality understands that good project engineering is too important to the enterprise and must be bullet-proof to these other considerations. As a result, gold-collar practitioners have processes and set ground rules that minimize the chance their projects will be derailed by considerations outside of the team's control. Too often, blue-collar firms are surprised by the failure of their new tool, cannot see what caused the failure, and reach the faulty conclusion that the tool does not work. Ineffective implementations are at the heart of a majority of tool adoption and process improvement failures [5].

Large-scale changes in the way we do business are often thwarted at the highest levels of leadership because the new paradigms are not compatible with the thinking and ideas that put our leaders where they are today.

From our position, the optimal approach is to chart a course toward becoming a gold-collar development firm leveraging gold-collar developers utilizing gold-collar processes. While we acknowledge the accomplishments of heroics, we see the long-term costs. Success comes from a wise combination of tactical skill and strategic wisdom. Championship pool is an excellent metaphor. It is not enough to have the skill needed just to make the next shot. True champions know where they want to leave the cue ball for the next shot as well as the rest of the shots on the table. Gold-collar workers are able to see the consequences of their actions and regularly leave themselves and their project teams better positioned for the next project than they were for the last. At the heart of a gold-collar firm is a focus on effectively leveraging the lessons others have learned. We believe an effective deployment of process is the method of choice to accomplish this.

As individual contributors, the choice should also be clear. The days of the solo developer coding a sequence of market-winning products are over. (That is assuming they ever existed.) The future is in high-performance teams of flexible gold-collar workers striving to bring real engineering to software development. As quoted above, a critical aspect of engineering is the "... application of scientific and empirical knowledge...." Being an engineer is not about what you know. It is how you go about applying what you and others know and what you can learn.

A 21st century engineer is more than someone with a set of skills, including process skills. These skills must be balanced with discipline and wisdom needed to be an effective member of a high-performance team. The Navy's elite flight demonstration team, the Blue Angels, is an excellent metaphor. A high-performance team does not happen overnight simply because a collection of experts is told it is a team. Building a team requires the right kinds of individuals and the right assembly process. Once again, poor implementations have given team-building exercises a bad name, but one can learn a great deal from the Blue Angels. Year after year, 50 percent of the team rotates and year after year the new team is formed. A critical aspect of the team's success is the time it spends becoming a team before its first public performance of the year [6]. If carefully selected team candidates are guided through truly meaningful team-building exercises, it is reasonable to assume the results might be as stunning as those of the Blue Angels.

Implementation Success

The concept of high-performance teams is not new or unproven. There are numerous examples from recent United States history where highly disciplined teams consisting of thousands of engineers not only undertook what were considered impossible goals, but succeeded. The Manhattan Project produced the nuclear bomb. The teams at NASA not only produced the Apollo series of space missions that put a man on the moon in 10 years, but also produced the incomparable Shuttle program. An oft-cited group, Lockheed's Skunkworks Division, produced the SR-71 Blackbird, a feat of technology that was three generations ahead of its day. In many ways, most of the military technology today is a result of high-performance teams leveraging state-of-the-art technology in ways previously unimagined. We believe it is our next task to make the creation of these teams and their successes more commonplace and at the basic level of all systems engineering, not just for high-value military systems.

There has been progress in software. Many technology consulting firms also possess high-performance teams aimed at improving the state of systems or enterprise engineering. Andersen, Deloitte & Touche, and Coopers & Lybrand, as well as other look-a-like companies have created their own brand of enterprise-engineering services. The Air Force also has had its share of success developing organic high-performance gold-collar teams. The point is that these teams already exist, as well as the process for creating them. Three examples help demonstrate our assertions.

Software Process Improvement

In the early 1990s, when the need for military software and software resources underwent a huge expansion, the Air Force made a deliberate decision to pursue excellence in the way it developed and maintained software. What followed was almost a decade of cooperation with the Software Engineering Institute in order to improve the state of Air Force software development. The CMM for software essentially was institutionalized across all Air Force software houses as well as within the contractor communities supporting the Air Force. In 1999, even though philosophical changes forced most of the Air Force away from primarily organic software efforts, organizations still controlling their own software projects retain a strong recognition and desire to use mature development techniques and processes as defined by the CMM. Such institutionalization is a result of the early commitment the Air Force made to create lasting results via a high-performing Software Process Improvement (SPI) team. There was a commitment to provide the people and resources necessary to develop critical skills and disciplines, build the programs necessary to take the fight to the units in the field, fund the temporary duties needed to work face-to-face with commanders, and stay the course in the face of early efforts that appeared, on the surface, to be failures. As results showed progress, the SPI team regularly evaluated options for improvement, with new capabilities and services continually added to their toolbox. The result has been impressive. Even though the SPI team officially disbanded in 1997, the cultural change within the Air Force software community has been surprising. At the height of the SPI team's activities, the most common remark heard from commanders was that there was not enough time and resources to make SPI happen via CMM. Almost three years later, we continue to hear about current commanders — those who were O-1 to O-4 at the height of the SPI effort — who claim they cannot develop good software without the guiding principles provided by the CMM.

The future is in high-performance teams of flexible gold-collar workers striving to bring real engineering to software development.

Year 2000

When the Year 2000 (Y2K) problem emerged as a serious threat to Air Force systems, it became a top priority almost immediately within the Communications and Information communities. Again, the Air Force made a conscious decision that Y2K must be urgently addressed by the highest levels of leadership within the Air Force. The Air Force Y2K Program Management Office (PMO) was developed and patterned after the Air Force SPI team. (Some personnel were on both teams.) The Air Force Y2K effort would utilize the guiding principle of centralized Air Force management and decentralized execution at the field and organization levels. Therefore, the processes and tools the PMO developed were key, as they would serve as the standard baseline for all Y2K remediation efforts undertaken by units in the field at all the various levels within the hierarchy. It is interesting to note that as in the SPI example, the Air Force was out in front of the other Department of Defense components for the Y2K effort. Many of the products, processes, training, and tools created by the Air Force PMO were subsequently adopted and used by various other governmental organizations for use in their respective Y2K efforts. It is noteworthy that many of the Air Force PMO-developed processes and tools were based on the concepts and practices defined by CMM. As in SPI, the Y2K PMO team's primary role was to develop the best practices and then shepherd the rest of the Air Force in using practices through face-to-face interaction that would include training, consulting, inspecting, and feedback conferences.

Network Management

The emerging network installation and network management teams provide a final Air Force example. At a time when networks are proving to be a most valuable and critical resource, network performance is sporadic and less than optimal. The Air Force response once again has been to turn to a high-performance gold-collar team to ensure that all base networks work, and work well. The highly trained SCOPE Network teams specialize in network management disciplines and have been created to standardize and optimize the performance of all Air Force base networks. These gold-collar teams travel to all sites to perform fine-tuning of networks, ensure adequate security, improve operations, train local personnel, and share best practices. As in the previous two examples, they utilize a disciplined process as they perform their work. As the Air Force experts, they shepherd the training and development of all network management personnel through face-to-face interaction, which includes training, consulting, and feedback. The SCOPE Network effort has proven to be a highly effective use of gold-collar development teams and has the data to prove its success.

If we step back and examine these three examples with respect to the differences described earlier, we find some interesting commonalities. Each of the examples required:

- dramatic improvement in specialized functions
- changes of core attitudes toward the use of disciplined processes (in the first two, this required changes of priorities by command-level leadership)
- merging of disparate, and sometimes confrontational, consumer and producer groups into integrated interdisciplinary teams with common goals
- use of the latest scientific methodologies, data and statistics, risk management, best practices, lessons learned, and other applicable leading-edge technologies into standardized, predictable, and repeatable processes
- change of old patterns of thought that people can only employ one or two specialized skills

Conclusion

From experience, it appears that the successful application of high-performance gold-collar teams has less to do with the technology being implemented than it does with other factors. Deciding if and when a gold-collar team is required is a crucial task that should not be taken for granted and is at least as important as the implementation of such teams.

We offer the following considerations as key criteria for helping to determine when high-performance gold-collar teams are an appropriate enterprise strategy:

- when an enterprise has deemed a capability or asset to be at the core of its capability to compete and survive in the marketplace;
- when an enterprise action is a priority that must meet specific performance levels, meet schedule time lines, and/or meet cost constraints.

The key ingredient, crucial for success, is twofold. First, there must be an identifiable need that can be articulated to the senior decision-makers of the enterprise in a way that causes them to take some action outside of the status quo. Second, there must be a conscious decision that the enterprise will pursue the gold-collar strategy for the long run. Once the decision is made to pursue, the commitment to its end must be total. These strategies are no longer off-the-wall experiments to achieve extraordinary results. If the enterprise will make the long-term investment, the results will be there in the end.

In all of the examples cited here, each has a common, yet subtle, characteristic that cannot be overlooked. Each of the situations required a change in the mindset of the prevailing culture of the day. All of today's leaders grew up with a frame of reference that was generally two to three iterations or more behind today's. As a result, large-scale changes in the way we do business are often thwarted at the highest levels of leadership because the new paradigms are not compatible with the thinking and ideas that put our leaders where they are today. This single fact is a primary reason why the success of efforts like these are so dependent upon the ability to interact face-to-face and work collaboratively, in a process-disciplined manner, to implement the change. Human interaction and the ability to manage resistance to the effort is as key to the high-performance team as any other aspect.

Final Thought

If software is mission-critical, design an environment to ensure it works. The decision to outsource is easy, but the cost to outsource and to ensure mission-critical capabilities survive is not trivial. Firms that outsource should follow the Air Force's approach to outsourcing the development and manufacture of fighters and bombers. The best pilots the Air Force has are intimately involved in every step of the process. What is the equivalent in your organization for your software-intensive systems? Gold-collar firms know when it is appropriate to outsource and they bring the same discipline to that task as they do to all the others. It is time for us all to make some critical choices and make the future a reality.

About the Authors

Lynn Robert Carter is a senior member of the technical staff of the Software Engineering Institute. From his office in Phoenix, Ariz., he supports the adoption of technology at a number of customer sites as a vehicle to learn how technology adoption can be made more predictable and less costly. Carter's focus is currently on the roles and responsibilities project leaders, managers, and executives must honor in order to establish environments conducive to the adoption of new technology without sacrificing mission capability. Carter has served as the director of Systems Engineering at EdgCore Technology, as president and CEO of Network Solutions, as a researcher at Motorola, and as a principal engineer at Tektronix. Carter served as an officer of the IFIP Working Group (2.4) on System Implementation Languages for 14 years, has published numerous papers, and has written and co-written three books. He received his bachelor's and master's degrees in mathematics from Portland State University and holds a doctorate in computer science from the University of Colorado at Boulder.

Software Engineering Institute
Carnegie Mellon University
3857 East Equestrian Trail
Phoenix, Ariz. 85044-3008

Voice: 480-598-1247
Fax: 480-496-9464
E-mail: lrc@sei.cmu.edu

Lt. Col. Scott Dufaud is deputy program manager for the U.S. Air Force Year 2000 Program Management Office at the Air Force Communications Agency (AFCA), Scott AFB, Ill. Prior to assuming these duties in November 1996, he was the chief of the Software Management Division at AFCA. Dufaud specializes in software management issues, software engineering process groups, software process improvement via the Capability Maturity Model, technology insertion, and issues of accelerating organizational change. He previously served at Headquarters Strategic Air Command and U.S. Strategic Command, the Air Force Manpower and Personnel Center, and Headquarters Air Force Space Command. He has a bachelor's degree in computer science from Southwest Texas State University and a master's degree in systems management from the University of Southern California.

HQ AFCA/ITY
203 W. Losey St, Rm 1065
Scott AFB, Ill. 62225-5224
Voice: 618-256-5697 DSN 576-5697
Fax: 618-256-2874 DSN 576-2874
E-mail: scott.dufaud@scott.af.mil

References

1. Kirby, Richard Shelton, et. al., *Engineering in History*, Dover Publications, New York, 1990.
2. Florman, Samuel C, *The Existential Pleasures of Engineering*, Second Edition, St. Martin's Griffin, New York, 1996.
3. Deming, W. Edwards, "Out of the Crisis," M.I.T. Center for Advanced Engineering Study, Cambridge, Mass. 1991.
4. Raymond, Erics S., editor, *The New Hacker's Dictionary*, The MIT Press, Cambridge, Mass. 1991.
5. Rummler, Geary A., Alan P. Brache, *Improving Performance*, Jossey-Bass Publishers, San Francisco, Calif., 1995.
6. Rear Adm. Moneymaker, Patrick D., personal conversations with a former commander of the Blue Angels.

Coming Events

16th International Conference on Data Engineering
Dates: Feb. 28-March 3, 2000
Location: San Diego, Calif.
Sponsor: IEEE Computer Society
Web site: http://www.research.microsoft.com/icde2000/

13th Conference on Software Engineering Education and Training
Theme: Software Engineering Coming of Age
Dates: March 6-8, 2000
Location: Austin, Texas
Sponsor: IEEE Computer Society
Web site: http://www.se.cs.ttu.edu/CSEET2000/confhome.htm

12th Software Engineering Process Group Conference (SEPG 2K)
Theme: 2000 Ways to Make Software Better
Dates: March 20-23, 2000
Location: Seattle, Wash.
Sponsor: Software Engineering Institute
Web site: http://www.sei.cmu.edu/products/events/sepg/

FOSE Leading-Edge Technology for Leaders in Government
Dates: Apr. 18-20, 2000
Location: Washington D.C.
Topic: FOSE offers a variety of educational opportunities for all levels of IT professionals.
Web site: http://www.fedimaging.com/conferences/

12th Annual Software Technology Conference
Theme: Software and Systems — Managing Risk, Complexity, Compatibility, and Change
Dates: Apr. 30-May 4, 2000
Location: Salt Lake City, Utah
Co-Sponsor: U.S Air Force, U.S. Army, U.S. Navy, U.S. Marine Corps, Defense Information Systems Agency, Utah State University Extension
Contact: Dana Dovenbarger
Voice: 801-777-7411
Fax: 801-775-4932
E-mail: dana.dovenbarger@hill.af.mil

First Software Product Line Conference (SPLC1)
Dates: Aug. 28-31, 2000
Location: Denver, Colo.
Sponsor: Software Engineering Institute
Web site: http://www.sei.cmu.edu/plp/conf/SPLC.html

23rd International Conference on Software Engineering
Dates: May 12-19, 2001
Location: Toronto, Ontario, Canada
Sponsor: IEEE Computer Society TCSE and Association for Computing Machinery SIGSOFT
Web site: http://www.csr.uvic.ca/icse2001/

Software Standards: Their Evolution and Current State

Reed Sorensen
TRW

This article touches on the last 30 years of software standards development in the Department of Defense, the purpose of and the difference between formal and de facto standards, the Ada experience, some current thinking regarding the documentation monster, and an update on J-STD-016 status.

"Stuff Happens" – so do Standards

If you sit down and try to categorize all the types of standards, the list gets really long really fast [1]. Under each type you start to come up with multiple entries. Standards seem to proliferate and infiltrate any aspect of human endeavor that involves any degree of complexity. Standards are not inherently good or bad, they just seem to inherently "be" — meaning that whether they are developed formally by a big, slow-moving international committee or whether they grow spontaneously from an innovative free market, standards are not going away.

While standards as a life-form will survive, evolving a specific species can be (and probably always is) difficult. Standards have to be cared for, they have to be fed, they have to evolve or they become extinct. Two examples:

1. Something as basic to the military as the ability of joint task forces to share standardized location data, just "ain't" that easy [2];
2. J-STD-016 has been going through the revision and ballot process for more than a year and should be undergoing re-balloting as you read this, but delays in this process have threatened its existence.

Software Standards Evolved Over 30 Years

Table 1 is a representative list of standards. Common school of thought has software standards evolving from the black ooze of hardware standards of the 1970s to early '80s. Some hardware standards began including words, sometimes in the form of appendices, to deal with software, e.g. MIL-STD-483 [3] and MIL-STD-490 [4]. Later, software-specific standards evolved that talked like software animals using software terminology, but walked liked hardware animals because they were direct hardware descendants, e.g. DOD-STD-2167 [5]. Today, the software community has troubled over the differences between software and hardware enough that software standards (EIA/IEEE J-STD-016 [6], IEEE/EIA 12207 [7]) are genuine software standards rather than feathered lizards.

Both MIL-STD-498 and J-STD-016 promote a more flexible approach to defining the software process and collaboration among all stakeholders, e.g. with joint technical and management reviews. Compared to older standards, MIL-STD-498 and J-STD-016 provide:

- a more complete life cycle perspective with links to system engineering
- better consistency with modern development models (incremental, evolutionary, reuse/re-engineering)
- improved accommodation of nonhierarchical design methods
- more flexibility with documentation formats and media
- attention to reusability, security, safety, and risk management
- improved references to metrics and indicators
- more responsive in-process evaluation and review activities
- significantly improved transition to software support and sustainment activities

Meanwhile, IEEE/EIA 12207 provides a higher-level view, with fewer specific requirements, and spans not only development, but acquisition, supply, operation, and maintenance.

Table 1. *Chronological list of some major software and software related standards.*

Date	Designator	Topic
1968	MIL-STD-490	Specification Practices
1970	MIL-STD-483	Configuration Management Practices
1978	DOD-STD-480A	Configuration Control
1979	MIL-S-52779A	Quality
1983	MIL-STD-1815A	Ada83 Language Reference Manual
1983	DOD-STD-7935	Documentation Standards
1983	DOD-STD-1679A	Software Development
1984	DOD-STD-1644B	Trainer System Software Development
1985	MIL-STD-490A	Specification Practices
1985	MIL-STD-1521B	Technical Reviews and Audits
1985	DOD-STD-2167	Defense System Software Development
1988	DOD-STD-7935A	Documentation Standards
1988	DOD-STD-2167A	Defense System Software Development
1988	DOD-STD-2168	Defense System Software Quality
1992	MIL-STD-973	Configuration Management
1994	MIL SPEC REFORM	
1994	MIL-STD-498	Software Development and Documentation
1995	J-STD-016-1995	Software Life Cycle Processes, Software Development, Acquirer-Supplier Agreement
1998	IEEE/EIA 12207	Software Life Cycle Processes
1998	EIA-649	Configuration Management
1999	J-STD-016	Software Life Cycle Processes, Software Development

Standards Enable Communication

Standards are about communication — communication between organizations, e.g. between the acquirer and developer, among developers, between the developer and the maintainer, communication between two commercial-off-the-shelf software products or two software units, and communication between a fax machine built by Canon and another built by Sharp.
The revolution in knowledge that accompanied the printing press was a communication revolution. The printing press provided the means to build upon the experience of those who came before[1]. A standard is either a specialized document that captures the experience of the many for the use of those who follow, or it is a groundswell that, by virtue of inertia, establishes default experience. In the first case, many may meet in committee and have a balloting process to gather experience from the broadest base of useful knowledge. In the second, a company may flood the marketplace with its solution to a problem in hopes that it will become a de facto standard — not that the solution is inherently the best but if the solution becomes omnipresent it will provide the most effective mechanism for most people to deal with most others (Microsoft Windows 3.1 for example).

Formal Standards Influence Software Development

Entire cottage industries pop up around formal standards. There are a lot of consulting organizations offering CMM® and ISO 9000[2] expertise. Seminars are available and tools are marketed to support "498" [8] and "12207." All of these kinds of free market ventures do their best to influence software development.

But not everyone buys into the concept of formal standards. Formal standards tend to be imposed by management rather than because the programmers sign a petition demanding that they want to be CMM Level 3, though life for the programmer is often better at Level 3 than Level 1.

The Interplay of Formal Standards, De Facto Standards, and Best Practices

Alex Polack of ATA, a vendor of software documentation technology, sees standards as a way to canonize best practices. Whether canonized or not, some advocates for best practices see them as a more natural and useful approach to improving software development than are formal standards. In *Rise and Resurrection* [9], Edward Yourdon suggests:

"If the standards/methods procedures group wants to rename itself as the best practices group, then it should realize that its job is to act like social scientists visiting an alien race deep in the forest: observing behavior, documenting existing practices, and offering advice on which practices have been observed to produce good results, e.g. higher levels of quality."

In concert with evolving best practices, the more recent high-level commercial standards, such as IEEE/EIA 12207, take a process-oriented approach and expect a software development organization to define and continuously improve its software process, then customize that process for individual development projects. The Department of Defense (DoD) Single Process Initiative further encourages developers to define and utilize its own processes.

A grassroots approach to standards is observed in the development tools, platforms, and graphical user interfaces that proliferate in the marketplace. Resulting de facto standards obviously influence development. A software development organization may be a "Windows shop" or a "Unix shop" or "Linux shop."

The Influence of the Ada Experiment

The Ada experiment caused some to step back and consider the purpose of requirements and design. You do not just sit down and start coding in Ada. It takes considerable work to get to a piece of Ada code that can be compiled, so you end up thinking a lot about the design. This thinking is indispensable in large systems.

Systems developed in Ada are arguably easier to maintain than had they been developed in C or another language less verbose than Ada. Because Ada is verbose it is easier for a maintainer to read than many languages, such as C, which are designed to be easy to write rather than easy to read. Ada is designed to be easy to read.

Did Ada influence the object-oriented community? While it is hard to measure such an influence, clearly Ada was the first language that enforced rather than simply permitted encapsulation and abstraction. You can argue that Java was influenced by Ada by observing that Java is C++ syntax with Ada semantics. Ada's contributions to software engineering aside, it was not a public relations success for the standards community. The 1986 mandate that Ada be used in new defense systems left a bad taste for many software development organizations, an aftertaste that remained even when free Ada compilers, trained Ada programmers, and Ada development tools became plentiful sometime later.

Data Item Descriptions and the "D" Word

With the announcement in 1994 by Secretary of Defense William Perry on Mil Spec Reform [10], the pendulum of combined thought had reached the zenith of its swing regarding the enforcement of formal standards. To a large extent, DoD got out of the expensive business of caring for and feeding standards. As the pendulum has swung from rigid enforcement of standards to total lack of enforcement[3], Data Item Descriptions (DIDs) often disappear from the contractual setting. This leaves many developers confused about how the data should be presented. Acquirer and developer find themselves reinventing the wheel. It is a little like discarding all known languages and developing your own, even though both parties speak English fluently. Like English, the DIDs provided a basis for communication. Time need not be used reinventing nor rediscovering, if acquirer and developer agree from the start to follow the J-

The Capability Maturity Model and CMM are registered in the U.S. Patent and Trademark office to Carnegie Mellon University.

STD-016 DIDs (called Software Product Descriptions or SPDs).

Often, the misapplication of 2167A resulted in documentation — the "D" word — that took on a life of its own. Whereas the software standards evolved from the primordial goo at the fringe of the hardware disciplines, the emergence of the associated software documentation was seemingly from a lab in Dr. Frankenstein's basement. It escaped to terrorize the DoD village, consuming lots of dollars, and causing general havoc among the defense populace. In an attempt to drive a wooden stake through the documentation monster's heart, no DIDs appear in IEEE/EIA 12207[4].

The apparent lack of DIDs has been/is troublesome for acquirers and developers who were accustomed to the detailed guidance provided by DoD-STD-2167A and even by MIL-STD-498. Much like a communist society without an iron curtain, the software acquisition/development community is confused by the new-found freedom of no DIDs. The developer is directed to provide "data," the popular euphemism for the "D" word. But what is this data to look like? Developers often want an example so they do not have to start from scratch. While IEEE/EIA 12207 does not leave one to start from scratch, some developers almost feel that way. Tables 3 and 4 show samples of the level of guidance provided by IEEE/EIA 12207. But the solution is available; one can use the materials referenced by IEEE/EIA 12207.1-1997 "Table 1 - Information item matrix," which includes the J-STD-016 SPDs and other IEEE standards or one can use J-STD-016, as shown in Table 2. Further, while MIL-STD-498 was cancelled in May 1998, the DIDs from MIL-STD-498 have not been cancelled, although they may be cancelled when the full use J-STD-016 is available.

The latest thinking on documentation is to use the natural work products of the development process for today's complex systems that are likely to need continuing evolution. Rather than requiring the developer to add extra steps to the development processes to transform software development products into a prescribed standard format, the developer may generate the required information directly from the development environment — as long as the information is usable by all stakeholders during initial and continued development and sustainment. Thus, documentation may be in the form of a requirements database, Computer-Aided Software Engineering (CASE) tool data, software development folders, and other artifacts produced during software development. For these reasons, EIA/IEEE J-STD-016 and MIL-STD-498 relaxed the requirements for documentation form to concentrate on usable content, i.e. capturing engineering and planning information in the form used for project management and development activities.

Useable Replacements	Obsolete DIDS		
J-STD-016 & MIL-STD-498 22 Product Descriptions Titles & MIL-STD-498 DID#s	**DOD-STD-2167A 17 DIDs Titles & DID#s**	**DOD-STD-7935A 11 DIDs Titles & DID#s**	**DIDs Superceded by MIL-498 DIDs**
Computer Programming Manual (CPM) DI-IPSC-81447	Software Programmer's Manual DI-MCCR-80021A		MCCR-80021A
Data Base Design Description (DBDD) DI-IPSC-81437		Database Specification DI-IPSC-80692	IPSC-80692; MCCR-80305
Interface Design Description (IDD) DI-IPSC-81436	IDD DI-MCCR-80027A	interface design from Unit Specification (US) DI-IPSC-80691	MCCR-80027A
Interface Requirements Specification (IRS) DI-IPSC-81434	IRS DI-MCCR-80026A	interface requirements from US DI-IPSC-80691	MCCR-80026A, 80303
Operational Concept Description (OCD) DI-IPSC-81430	systems concept from SSDD	systems concept from Functional Description (FD) DI-IPSC-80689	IPSC-80689
Software Design Description (SDD) DI-IPSC-81435	SDD DI-MCCR-80012A	design from US, MM	MCCR-80012A, 80304, 80306; IPSC-80691
Software Development Plan (SDP) DI-IPSC-81427	SDP DI-MCCR-80030A	section 7 of FD	MCCR-80030A, 80297, 80298, 80299, 80300, 80319
Software Product Specification (SPS) DI-IPSC-81441	SPS DI-MCCR-80029A plus maintenance programmers' procedures from CRISD	Maintenance Manual (MM) DI-IPSC-80696	MCCR-80029A, 80317; IPSC-80696
Software Requirements Specification (SRS) DI-IPSC-81433	SRS DI-MCCR-80025A	requirements from Unit Specification DI-IPSC-80691	MCCR-80025A, 80301
System/Subsystem Design Description (SSDD) DI-IPSC-81432	S/Segment DD DI-CMAN-80534 (without OCD data)	design data from FD and System/ Subsystem Spec (SS) DI-IPSC-80690	CMAN-80534; MCCR-80302
Software Transition Plan (STrP) DI-IPSC-81429	Computer Resources Integrated Support Document (CRISD) DI-MCCR-80024A (without maintenance programmers' procedures)	MM planning data	MCCR-80024A
System/Subsystem Specification (SSS) DI-IPSC-81431	S/Segment S DI-CMAN-80008A	requirements from FD, SS	CMAN-80008A; IPSC-80690
Software Test Description (STD) DI-IPSC-81439	STD DI-MCCR-80015A	detailed portion of Test Plan (PT) DI-IPSC-80697	MCCR-80015A, 80310
Software Test Plan (STP) DI-IPSC-81438	STP DI-MCCR-80014A	high level portion of PT	IPSC-80697; MCCR-80014A, 80307, 80308, 80309
Software Test Report (STR) DI-IPSC-81440	STR DI-MCCR-80017A	Test Analysis Report DI-IPSC-80698	MCCR-80017A, 80311; IPSC-80698
Software User Manual (SUM) DI-IPSC-81443	SUM DI-MCCR-80019A	End User Manual DI-IPSC-80694	MCCR-80019A, 80313, 80314, 80315; IPSC-80694
Software Version Description (SVD) DI-IPSC-81442	Version Description Document DI-MCCR-80013A		MCCR-80013A, 80312

Table 2. *A partial list of obsolete DIDs and usable replacements, excerpted from* Army Communications-Electronics Command Common Equivalent (ECOM) Software Engineering Center Request for Proposal Guidebook (www.sed. monmouth.army.mil/se), *Operations, Strategic Planning & Policy, RFP Guide.*

Are Current Software Standards Used?

Yes. Gary Hebert, a civilian electronics engineer at Hill AFB, says he used IEEE/EIA 12207 to understand the current thinking on documentation. He had come from a 2167A background. He found that IEEE 12207 provided flexibility allowing the use of electronic documents rather than hardcopy. Based on this direction, his organization uses Ives Development's "Team Studio Analyzer" to capture the design of a Lotus Notes database system. In this case, the tool generates a record describing the attributes for each of the Lotus Notes components, e.g. forms, views, subforms, agents, scripts or subroutines. These records are part of a documentation database. According to Nigel Cheshire, CEO for Ives Development, one of the advantages of a database over a traditional collection of documents is the ability to generate reports against the database to verify that the software design complies with an organization's standards. For example, you could generate a report to verify that each user-editable field has help text attached as per your internal corporate standard.

And no. In speaking with some Air Force personnel, I find that early this year they were using MIL-STD-498 as a reference document to recommend a series of software development documents and also to develop a concept of operations using the Operational Concept Description Data Item Description (DID). They were just barely aware of the commercial version of the standard (J-STD-016) and were unaware of IEEE/EIA 12207.

An Update on J-STD-016

While MIL-STD-498 was cancelled in May 1998, the J-STD-016 is still alive and well and is being updated. Those interested can get the trial-use J-STD-016-1995 now, and will be able to get the updated J-STD-016 when it is available. Both the trial-use version and the full-use version will provide that same reference functionality they were getting from MIL-STD-498, but with the benefit of the latest thinking.

With the DoD prohibition of putting standards on contract, the provisions in J-STD-016 for contractual use will be changed to provide for use as a basis of agreement on the activities and work products of the development process, where the agreement may take any form, from a handshake to a formal contract, and may be within an organization or between organizations. The full-use standard will be intended for project-specific application, on legacy systems and in organizations that have legacy processes (i.e., processes based on the old DoD standards), not as a basis of defining an organizational life cycle process. The expectation is that defense and commercial developers and acquirers will use international and professional standards such as IEEE/EIA 12207 as a basis for their long-term organizational process definition.

The conversion of J-STD-016 to a full-use standard was

Purpose: Describe a planned or actual function, design, performance, or process.

A description should include:

a) date of issue and status
b) scope
c) issuing organization
d) references
e) context
f) notation for description
g) body
h) summary
i) glossary
j) change history

Table 3. *Description — generic content guideline from IEEE/EIA 12207.*

Table 4. *Description — software interface design from IEEE/EIA 12207.*

Purpose: Describe the interface characteristics of one or more system, subsystem, hardware item, software item, manual operation, or other system component. May describe any number of interfaces.

IEEE/EIA 12207.0 reference 5.3.5.2, 5.3.6.2

Content: The software item interface design description should include:

a) generic description information (See Table 3)
b) external interface identification
c) software component identification
d) software unit identification
e) external-software item interface definition (e.g., source language, diagrams)
f) software item-software item interface definition (e.g. source language, diagrams)
g) software component-software component interface definition (e.g., source language, diagrams)

Characteristics: The software item interface design description should:

a) support the life cycle data characteristics from annex H of IEEE/EIA 12207.0 (see 4.2 of this guide)
b) define types of errors not specified in the software requirements and the handling of those errors

delayed in Environmental Impact Analysis (EIA) balloting and has been further delayed in combined EIA and IEEE ballot resolution due to a lack of resources to complete the resultant revisions. Hopefully, by the time this article is published, re-balloting will be in progress and the schedule for its completion will be available from the EIA and the IEEE.

EIA/IEEE J-STD-016 and MIL-STD-498 DIDs vs. DOD-STD-2167A and DOD-STD-7935A

Table 2 provides usable product descriptions from J-STD-016 and DIDs from MIL-STD-498. The DIDs from DOD-STD-2167A and DOD-STD-7935A were harmonized to produce the DIDs of MIL-STD-498. The product descriptions in the annexes of J-STD-016 are the commercial equivalents of the DIDs in MIL-STD-498, and have the same names.

For each obsolete DID associated with DOD-STD-2167A and 7935A, the table shows the corresponding usable J-STD-016 product description and MIL-STD-498 DID. The table also shows the DIDs that each MIL-STD-498 DID officially superceded; some were from DOD-STD-2167A or 7935A and some were from other MIL-STDs.

(Note that 498 has been cancelled, but as of this writing, the DIDs have not been cancelled.)

Acknowledgements

I want to thank Marilyn Ginsberg-Finner for providing her comments, the J-STD-016 updates, and Table 2. Les Dupaix, Dr. David Cook, Gary Hebert, and Alex Polack, also contributed to this article.

About the Author

Reed Sorensen has more than 20 years experience with TRW developing and maintaining software and documentation of embedded and management information systems; providing systems engineering and technical assistance to program offices, and consulting with many DoD organizations regarding their software configuration management and documentation needs. He provides configuration management, software control, interface control, and systems requirements analysis in support of intercontinental ballistic missile sustainment, modifications, and replacement programs. Sorensen has published several articles in *CrossTalk* on various software related subjects.

TRW ICBM Systems
Attn: Reed Sorensen N14GC
888 South 2000 East
Clearfield, Utah 84015-6216
Voice: 801-525-3357
Fax: 801-525-3355
E-mail: Reed.Sorensen@siinet.trw.com

References

1. Some sources of lists — http://www-library.itsi.disa.mil./, http://standards.ieee.org/catalog/index.html, http://global.ihs.com/cgi-bin/litmus_test.cgi?FRITTER=134547&NODE=PC
2. Polydys, M.L., "Operation Data Storm: Winning the Interoperability War through Data Element Standardization," *CrossTalk*, July 1999.
3. MIL-STD-483, Configuration Management Practices, 1970.
4. MIL-STD-490, Specification Practices, 1968.
5. DOD-STD-2167, Defense System Software Development, 1985.
6. J-STD-016, Software Lifecycle Processes, 1999.
7. IEEE/EIA 12207, Industry Implementation of International Standard ISO/IEC 12207:1995 (ISO/IEC) Standard for Information Technology, 1998.
8. MIL-STD-498, Software Development and Documentation, 1994.
9. Yourdon, Edward, *Rise and Resurrection of the American Programmer*, Yourdon Press Computing Series, page 128.
10. Perry,William J., Secretary of Defense, DoD Policy on the Future of MILSPEC, *CrossTalk* September 1994.

Notes

1. This came when the advent of written language by the press made it available on a scale of a magnitude greater than handwriting provided.
2. A series of international standards on quality.
3. Alex Polack notes that from the mid-1980s to the mid-'90s, there was a tendency to use standards as a club to beat up or monitor developers. Marilyn Ginsberg-Finner notes that within current DoD acquisition reform policy, standards are primarily guidance to developers in defining their software process and standards are still essential as a basis for acquirers to evaluate the processes, activities, and work products that an offeror proposes and provides.
4. IEEE/EIA 12207.1-1997 "Table 1 - Information item matrix" references standards, including J-STD-016 that have DID-like guidance.

Influential Men and Women of Software

Kathy Gurchiek
CrossTalk Managing Editor

Looking at the Evolution of Software would not be complete without a nod to those software heavyweights whose contributions furthered software development. Below, CrossTalk *recognizes some of those people.*

Charles Babbage

Inventor in the early 1800s of the difference engine and design for the analytical engine. He was a well-educated young man in Great Britain who wanted to be a mathematician and scientist. One evening in 1812 while studying a table of logarithms, he realized calculating them could be automated and he began to design the Analytical Engine for that purpose. Like projects since, it was never completed due to cost overruns and reluctance by the British to provide additional funding.

Lady Ada Augusta Byron

Countess of Lovelace and daughter of poet Lord Byron, she was a 19th century mathematician who has unofficially been called the first computer programmer. Through lengthy correspondence and notes during the 1800s, she interpreted Babbage's "thinking machine," and was able to foresee, and describe in simple terms, the symbolic processing by machine.

"It would appear that we have reached the limits of what it is possible to achieve with computer technology, although one should be careful with such statements, as they tend to sound pretty silly in five years."
— John Von Neumann, 1949

Alan Kay

Credited with coining the term "object-oriented" (OO) programming language, a term which came into use in 1970, four years after the first OO programming language (Simula) was introduced.

Alan Turing

In the 1930s he developed what is now known as the Turing Machine, a theoretical computer that could "compute all that is computable" using limited instructions and infinite working storage. Using it as standard, two Italian mathematicians proved that any programming language needed only the sequence, the decision statement, and the iterative looping structure to implement any computable algorithm.

Software Time Line

1850 — George Boole represents logic states with zeros, numbers

1940 — Claude Shannon recognizes relationship between Boolean logic and electronic circuits

1947 — Howard Aiken, part of an IBM-backed team at Harvard, predicts that the United States will need a total of six electronic digital computers.

1950 — Adm. Grace Murray Hopper invents the modern concept of the compiler and is instrumental in Common Business-Oriented Language (COBOL).

Late 1950s — FORTRAN (FORmula TRANslator) is born, designed by John Backus for IBM. Oldest high-level programming language.

1963 — Douglas Engelbart, working at Stanford Research Institute, receives a patent on the mouse; he demonstrates his keyboard, keypad, mouse, and windows system in 1968.

1965 — Ted Nelson introduces concept of hypertext, the language of the World Wide Web and an alternative concept in database design.

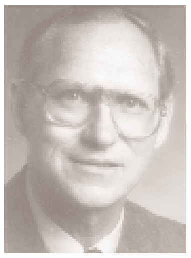
1969 — Internet is born when Leonard Kleinrock, a professor at the University of California at Los Angeles, gives the order to send the first message over the Net. After typing in "log," the system crashed.

Capers Jones

Known for work in the quantification of software productivity and quality, and in developing estimation and measurement tools, which use his quantification methods. In the early 1970s, he noticed there was a consistent bias in software project data. The discovery of bias patterns in lines of code metric confirmed the economic validity of Allan Albrecht's function point metric. His book, *Programming Productivity*, was the first to quantify the paradox of lines of code metrics and show the bias by language level.

Watts S. Humphrey

Research scientist for the Software Process Program he founded as part of the Software Engineering Institute located in Pittsburgh. He is the author of *A Discipline for Software Engineering* (1995), *Managing Technical People* (1996), and *Introduction to the Personal Software Process* (1997).



Barry W. Boehm

While employed by TRW Corp., Boehm wrote *Software Engineering Economics*. Published in 1981, it completely described the Constructive Cost Model (COCOMO) used in software cost and schedule estimation. With his book, this model quickly gained popularity and user's groups emerged throughout the world.

Tim Berners-Lee

Inventor of the World Wide Web, which runs on the Internet, and hailed by *Time* magazine as one of the 100 greatest minds of this century. The web made the Internet useable and useful to all types of people and applications. Before the web was born, the Internet was used mostly by scientists and the military and one had to be a progammer to make use of it. He is the director of the World Wide Web Consortium at the MIT Laboratory for Computer Science.

Edward Yourdon

The publisher of *American Programmer* is widely known as the developer of the method of structured systems analysis and design.

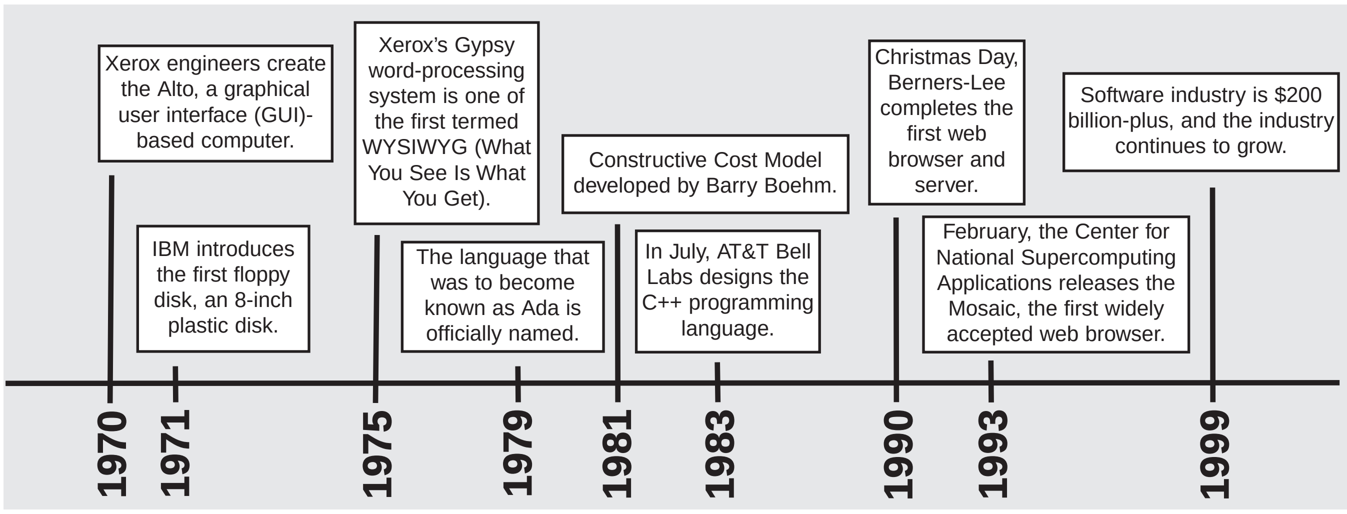
"Technology makes it possible for people to gain control over everything, except technology."
— John Tudor

Adm. "Amazing" Grace Murray Hopper

Creator of Common Business Oriented Language (COBOL). She was an officer in the Navy who became an Admiral. COBOL came about in the 1950s when the need for higher order languages was seen as a way to increase the productivity of programming computer applications.

References:

1. Marciniak, John J., Editor-in-Chief, *Encyclopedia of Software Engineering*, John Wiley & Sons Inc., New York, 1994, pages 25, 36, 103, 408, 605, 688-89, 691, 721.
2. World Wide Web History Project, http://www.webhistory.org/historyday/abstracts.html
3. Humphrey, Watts, *CrossTalk: The Journal of Defense Software Engineering*, February 1999
4. Keyes, Jessica, *Software Engineering Productivity Handbook*, McGraw Hill, 1992, pp. 80, 116, 332.
5. A Brief History of the Computer, http://www.qvctc.commnet.edu/classes/csc277/timeline.html


1970
Xerox engineers create the Alto, a graphical user interface (GUI)-based computer.
1971
IBM introduces the first floppy disk, an 8-inch plastic disk.
1975
Xerox's Gypsy word-processing system is one of the first termed WYSIWYG (What You See Is What You Get).
1979
The language that was to become known as Ada is officially named.
1981
Constructive Cost Model developed by Barry Boehm.
1983
In July, AT&T Bell Labs designs the C++ programming language.
1990
Christmas Day, Berners-Lee completes the first web browser and server.
1993
February, the Center for National Supercomputing Applications releases the Mosaic, the first widely accepted web browser.
1999
Software industry is $200 billion-plus, and the industry continues to grow.

Editor's note: *For a humorous look at the history of computer science, check out the time line at* http://baetzler.de/humor/hist_cs.html

CrossTalk Article Index: 1999

+ Articles marked with a plus sign appear only in the web edition of CrossTalk.

Article	Author	Month
Best Practices		
"Improving Software Engineering Practice"	P. Sanders	January
"High-Leverage Best Practices — What Hot Companies are Doing to Stay Ahead and How DoD Programs Can Benefit"	N. Brown	October
"16 Critical Software Practices for Performance-Based Management"	J. Lochner	October
"Software Mini-Assessments: Process and Practice"	G. Natwick, G. Draper, L. Bearden	October
"Experience in a Bottle: How Boeing Captured its Assessment Best Practices"	D. Corbin, R. Hamerly, R. Cox, K. Knight	October
"A Multi-Site Software Process Framework"	R. Porter Jr. D. DeToma	October
"A Problem-Based Approach to Software Process Improvement: A Case Study"	J. Rothman	October
"Best Practices — Presentation is as Important as the Issue"	G. Jackelen	October
CMM Level 5		
"The Capability Maturity Model: A Summary"	M. Paulk, et. al	May
"The Journey to CMM Level 5: A Time Line"	P. Cosgriff	May
"Benefits Realized from Climbing the CMM Ladder"	L. Oldham, D. Putman, M. Peterson, B. Rudd, K. Tjoland	May
"The Right Things for the Right Reasons"	P. Cosgriff	May
"Assessing a Level 5 Organization"	M. Paulk, D. Putman	May
Configuration Management		
"CCB — An Acronym for "Chocolate Chip Brownies"?	R. Sorensen	March
"Using CM to Recapture Baselines for Y2K Compliance Efforts"	R. Starbuck	March
"Configuration Management: Coming of Age in The Year 2000"	C. Burrows	March
Defense Information Infrastructure Common Operating Environment		
"Introduction to the DII COE Environment"	P. Engbert, J. Surer	September
"Extending the DII COE for Real-Time"	Lt. Col. Robillard, H. Rebecca Callison, J. Maurer	September
+ "Overview of the DII COE 4.0 Kernel"	S. Chubin, T. McVittie, R. Miller	September
Evolution of Software		
"Software Standards: Their Evolution and Current State"	R. Sorensen	December
"Evolution of Programming Languages and Why a Language is Not Enough to Solve Our Problems"	D. Cook	December
"21st Century Engineer"	L. Carter, S. Dufaud	December
"Influential Men and Women of Software"	K. Gurchiek	December
Interviews		
"Up Close with Dr. Etter"	K. Gurchiek	July
"Up Close with Gen. Lyles"	K. Gurchiek	December
Java		
"Real-World Java Development Experiences"	J. Soller, J. Clingenpeel, P. Hayes Jr., M. Muday, B. Larsen, T. Jones	January

Measures and Metrics

"An Effective Metrics Process Model"	Capt. T. Augustine, C. Schroeder	June
"PSM Insight: The Army-DoD Tool to Implement Issue-Driven Software Measurement"	D. Lucero	June
"It Is the People Who Count In Measurement: The Truth about Measurement Myths"	C. Dekkers, M. Bradley	June

Miscellaneous

"Software Product Lines"	P. Clements	February
"Design Maturity Model"	D. Przebowski	June
"Privatization of the Web: Hidden Economic Backhand"	J. Brower	June
"Operation Data Storm: Winning The Interoperability War through Data Element Standardization"	M. Polydys	July
"Determining the Completed Effort of Adapting Existing Software"	R. Cobb, D. Wood, L. Smeraglinolo	November
"Evolution of *CrossTalk*"	T. Stauder	December

Project Management

"Managing (the Size of) Your Projects"	C. Dekkers	February
"Applying Management Reserve To Software Project Management"	W. Lipke	March
"Gaining Confidence in Using Return On Investment and Earned Value"	L. Smith, A.Todd Steadman	April
"PAIR: A Rational Approach to Fighting Software Project Fires"	G. Daich	June
"Earned Value Project Management: An Introduction"	Q. Fleming, J. Koppelman	July
"Something from Nothing"	J. Smedra	July
"Managing Risk Management"	A. Neitzel Jr.	July

Requirements Management

"Writing Effective Natural Language Requirements Specifications"	W. Wilson	February
"Making Requirements Management Work For You"	A. Davis, D. Leffingwell	April
"Validating Software Requirements"	J. Sharp	November

Simulation

"Building Self-Reconfiguring Distributed Simulations Using Compensating Reconfiguration"	Lt. Col. Welch, J. Purtilo	January
"Simulation: An Enabling Technology In Software Engineering"	A. Christie	April

Software Acquisition

"Outsourcing and Privatizing Information Technology"	J. Brower	January
"Acquisition Software Oversight"	D. Etter	August
"Using Contractor Capability Evaluations to Reduce Software Development Risk"	B. Troup, B. Gallagher	August
"Effective Acquirer/Supplier Software Document Reviews"	G. Daich	August
"Product Line Acquisition in the DoD: The Promises, The Challenges"	L. Jones	August
"The Software Insight Tool: A Tool and Methodology for Risk Mitigation and CIO Assessments"	J. Kastning, J. Herman, M. Ginsberg-Finner, J. Heil	August
"Outsourcing Acquisition and Procurement Shops"	J. Brower	November

Software Process Improvement

"Using the TSP on the TaskView Project"	D. Webb, W. Humphrey	February
"The Rosetta Stone"	D. Reifer, B. Boehm, S. Chulani	February
"Sponsoring Process Improvement"	Col. H. Mason	April
"USDA's National Finance Center: A 10-Month Journey to Level 2	A. Todd Steadman	June
"Mission-Critical and Mission-Support Software: A Preliminary Maintenance Characterization"	E. Clark, J. Forbes, E. Baker, D. Hutcheson	June
"The Five Stages of Denial"	R. Bechtold	September

Software Quality Assurance

"Effective Software Defect Tracking"	B. Subramaniam	April
"Software Quality Assurance in a CMM Level 5 Organization"	R. Craig	May
"Confusing Process and Product: Why the Quality is not There Yet"	D. Cook	July
"Performing V&V in Architecture-Based Software Engineering"	E. Addy	September

Technology Change Management

+ "Managing Software Innovation And Technology Change Workshop"	E. Forrester, P. Fowler, S. Guenterberg	November
"Integrating Knowledge and Processes In the Learning Organization" [pt. 1]	L. Levine	November
"Structured Approaches to Managing Change"	M. Paulk	November
"Addressing People Issues when Developing and Implementing Engineering Processes"	C. Laporte, S. Trudel	November
"Rapid Emerging Knowledge Deployment"	K. Marler	November

Y2K

" Year 2000 Compliance 1999 Reporting Requirements"	A. Money	January
"The Network is Down…"	Capt. C. Walter	January
"Estimating Y2K Rework Requirements"	L. Fischman, P. McQuaid	January
"Effective Methods for Testing Year 2000 Compliance"	W. Perry	January
"The Upside of Y2K"	J. Hubbs	February
"Logical Event Contingency Planning for Y2K"	R. Moore, R. Krupit	March
"A Y2K Integration Test Model"	W. Dashiell	September
"Who is to Blame for the Y2K and Similar Bugs?"	A. Jarvis, V. Crandall, C. Snow	September
+ "Y2K Defect Propagation Risk Assessment using Achilles"	D. Welch, K. Greaney	November

BackTALK

How Far Have We Come?	D. Cook	December
What is the Buzz about Change?	G. Petersen	November
Who Knows Best?	G. Petersen	October
One Flew Over the Cuckoo's Cubicle	G. Petersen	September
Software and the 15th Century	G. Petersen	August
Farewell from Lorin May	L. May	July
The Acid Test: Measuring Your Success	G. Petersen	June
The Key to Effective Writing and Coding: Quality Assurance	L. May	May
Vinegar and Dye Pellets Not Included	L. May	April
Don't Forget the Feather Boa	L. May	March
Legacy of the Information Age	L. May	February
Need Proof? Call 1-900-Y2K-SHAM	L. May	January

From the Publisher

Endless Possibilities	R. Alder	December
CMMI Offers an Enterprise Focus for Teahnology Change Management	Lt. Col. J. Jarzombek	November
Practice Makes Perfect	T. Stauder	October
The Strategic Battlefield	R. Alder	September
Integrating Acquisition with Software and Systems Engineering	Lt. Col. J. Jarzombek	August
Sharpening Your Management Skills	T. Stauder	July
The Need for a Measurement and Analysis Process: Focusing on Guidance for Process Improvement	Lt. Col. J. Jarzombek	June
Process Maturity Pays Off in Many Ways	D. Wynn	May
Test Drive Your Software	T. Stauder	April
Who Needs John Wayne?	R. Alder	March
Software Knowledge Management-Strengthening Our Community of Practice	Lt. Col. J. Jarzombek	February
Short-Term Fixes Shade Future	F. Brown	January

CrossTALK

SPONSOR	*Lt. Col. Joe Jarzombek*
PUBLISHER	*Reuel S. Alder*
ASSOCIATE PUBLISHER	*Tracy Stauder*
MANAGING EDITOR	*Kathy Gurchiek*
ASSOCIATE EDITOR/LAYOUT	*Heather L. King*
ASSOCIATE EDITOR/FEATURES	*Heather Winward*
VOICE	801-775-5555 DSN 775-5555
FAX	801-777-8069 DSN 777-8069
E-MAIL	crosstalk.staff@hill.af.mil
STSC ONLINE	http://www.stsc.hill.af.mil
CROSSTALK ONLINE	http://www.stsc.hill.af.mil/ Crosstalk/crostalk.html
ESIP ONLINE	http://www.esip.hill.af.mil

Subscriptions: Send correspondence concerning subscriptions and changes of address to the following address:
Ogden ALC/TISE
7278 Fourth Street
Hill AFB, Utah 84056-5205
E-mail: stsc.custserv@hill.af.mil
Voice: 801-775-5555
Fax: 801-777-8069 DSN 777-8069

Editorial Matters: Correspondence concerning Letters to the Editor or other editorial matters should be sent to the same address listed above to the attention of *CROSSTALK Editor*.

Article Submissions: We welcome articles of interest to the defense software community. Articles must be approved by the *CROSSTALK* editorial board prior to publication. Please follow the *Guidelines for CROSSTALK Authors*, available upon request. We do not pay for submissions. Articles published in *CROSSTALK* remain the property of the authors and may be submitted to other publications.

Reprints and Permissions: Requests for reprints must be requested from the author or the copyright holder. Please coordinate your request with *CROSSTALK*.

Trademarks and Endorsements: This DoD journal is an authorized publication for members of the Department of Defense. Contents of *CROSSTALK* are not necessarily the official views of, or endorsed by, the government, the Department of Defense, or the Software Technology Support Center. All product names referenced in this issue are trademarks of their companies.

Coming Events: We often list conferences, seminars, symposiums, etc., that are of interest to our readers. There is no fee for this service, but we must receive the information at least 90 days before registration. Send an announcement to the CROSSTALK Editorial Department.

STSC Online Services: This can be reached on the Internet. World Wide Web access is at http://www.stsc.hill.af.mil. Call 801-777-7026 or DSN 777-7026 for assistance, or e-mail to schreifr@software.hill.af.mil.

Back Issues Available: The STSC sometimes has extra copies of back issues of *CROSSTALK* available free of charge. If you would like a copy of the printed edition of this or another issue of *CROSSTALK*, or would like to subscribe, please contact the customer service address listed above.

The Software Technology Support Center was established at Ogden Air Logistics Center (AFMC) by Headquarters U.S. Air Force to help Air Force software organizations identify, evaluate, and adopt technologies that will improve the quality of their software products, their efficiency in producing them, and their ability to accurately predict the cost and schedule of their delivery. *CROSSTALK* is assembled, printed, and distributed by the Defense Automated Printing Service, Hill AFB, Utah 84056. *CROSSTALK* is distributed without charge to individuals actively involved in the defense software development process.

How Far Have We Come?

To put the evolution of software in perspective, you need to understand the typical mentality of new programmers and coders fresh from college and technical school. It will really make you feel “mature” and “experienced” — both euphemisms for “old” — to understand that today’s new programmers . . .

- Think that seven- and nine-track tapes were variations of eight-track tapes. Somehow, the “Beach Boys” come to mind when you mention them.
- Think that Core Memory has to do with business values.
- Have never used a punch card, cannot tell you how many columns are on a punch card, and do not understand what a keypunch machine is.
- Have never saved the confetti-like bits of paper from a keypunch to use later. Of course, the correct use was to trash a friend’s desk or car. (The bits of paper were called chaff or chad. If you want to know why it was called chad, e-mail me at david.cook@hill.af.mil).
- Vaguely remember when floppy drives were floppy. If they remember “real” floppies, they think they were always 5¼ inches (remember 8-inch floppies?).
- Do not feel fear and revulsion when they hear the acronym JCL. They do not even know it means Job Control Language.
- Think that loading a word processor that consumes 40-plus megabytes of memory is no big deal.
- Think that leasing a machine means going to CompUSA and spending under $100 per month.
- Cannot remember not having a hard drive to copy files to. Remember copying files on a one-floppy machine? It was like playing the accordion to copy multiple files.
- Cannot comprehend editing a file without using a mouse — and if they have Unix experience, never used VI (visual information) and probably like EMACS!
- Have never coded in a language that required segmentation or overlays of code.
- Never used a calculator — or heaven forbid, a slide rule — to check the correctness of a FORTRAN program.
- Have never read — or probably even seen — a core dump, and have never had to calculate a physical vs. virtual address to debug.
- Think that ASCII (the American National Standard Code for Information Interchange) is the only character set, and have never heard of EBCDIC (Extended Binary Coded Decimal Interchange Code). (Remember? Nonsequential character sequences, multiple versions, and important characters missing).
- Have never waited hours — or days — for the results of a compile. (Because of this, they have never understood the importance of desk-checking code prior to a compile.)
- Cannot remember when compile-link-load-execute were separate steps in running a program. (Each step took a separate deck of JCL cards!)
- Think that “booting” a machine only requires one to turn on the power.
- Have always used WYSIWYG (What You See Is What You Get) word processors. If you did not already know that acronym, you are the youngster this column is aimed at.
- Never sweet-talked an operator into upping the priority of a job to get a quicker compile. (They probably do not understand what a computer operator really does.)
- Have become accustomed to using an operating system that crashes or hangs routinely, sometimes multiple times per day.

— David Cook, Software Technology Support Center

Got an idea for *BACKTALK*? Send an e-mail to crosstalk.staff@hill.af.mil

CROSSTALK
Ogden ALC/TISE
7278 Fourth Street
Hill AFB, UT 84056-5205

BULK RATE
US POSTAGE PAID
Permit No. 481
Cedarburg, WI