activity in federal service to achieve SEI CMM® Level 4 distinction. The TPS and IA functions, under his direction, became ISO 9001/TickIT registered in 1998. These functions were honored with the 1999 IEEE Computer Society Award for Software Process Achievement. Lipke is a professional engineer with a master's degree in physics.

Jeff Vaughn is the Metrics and Financial Analyst of the Oklahoma City Air Logistics Center's Software Division. He has 13 years experience in Avionics Test Program Set Development and Maintenance. He managed one of the first organizations in the Software Division to utilize EV Management techniques to manage projects. He has a bachelor's degree in electrical engineering.

OC-ALC/LAS
Tinker AFB, Okla. 73145-9144
Voice: 405-736-3335
Fax: 405-736-3345
E-mail: wlipke@lasmx.tinker.af.mil
jvaughn@lasmx.tinker.af.mil

Letter to the Editor

Dear CrossTalk:

Ever since I've been a CrossTalk reader, which has been around four to five years, I look forward to the next issue. CrossTalk is one of the best, if not the best, publications in the software process improvement (SPI) arena. There are always practical lessons learned in the wide variety of articles, something I have been able to use in my work. In addition, CrossTalk now includes Web links to many valuable sources of information.

As a former government employee and government contractor, I have been a frequent beneficiary of sharing information. Just because something was developed by the government does not mean it can not be used in private industry and vice versa. There is no sense in making the same mistakes. CrossTalk is a great help in sharing lessons learned in areas of value to our company—risk management, managing change, project management, and others related to SPI. Also, the CrossTalk staff has always been a pleasure to work with and has a positive, can do attitude. Thanks for helping those of us in the SPI trenches.

Darrell Corbin
The Boeing Company

Large Software Systems–Back to Basics

by John R. Evans
SPAWAR Systems Center

Our computer hardware is growing in power exponentially. We naturally expect to use this power on larger, more complicated problems. There is a problem, however. Software development methods that worked fine on small problems seem to not scale well.

The Problem and Partial Solution

Today when we launch a software project, its likelihood of success is inversely proportional to its size. The Standish Group reports that the probability of a successful software project is zero for projects costing $10 million or more [1]. This is because the complexity of the problem exceeds one person's ability to comprehend it. According to The Standish Group, small projects succeed because they reduce confusion, complexity, and cost. The solution to the problem of building large systems is to employ those same techniques that help small projects succeed—minimize complexity and emphasize clarity.

The goals, constraints, and operating environment of a large software system, along with its high-level functional specification, describe the requirements of the systems. Assuming we have good requirements, we can decompose our system into smaller subsystems. Decomposition proceeds until we have discrete, coherent modules. The modules should be understandable apart from the system and represent a single idea or concept. When decomposition is finished, the modules can be incorporated into an architecture.

Frederick P. Brooks said that the conceptual integrity of the architecture is the most important factor in obtaining a robust system [2]. Brooks observed that it can only be achieved by one mind, or a very small number of resonant minds. He also made the point that architectural design must be separated from development. In his view, a competent software architect is a prerequisite to building a robust system.

An architecture is basically the framework of the system, detailing interconnections, expected behaviors, and overall control mechanisms. If done right, it lets the developers concentrate on specific module implementations by freeing them of the need to design and implement these interconnections, data flow routines, access synchronization mechanisms, and other system functions. Developers typically expend a considerable amount of energy on these tasks, so not doing them is a considerable savings of time and effort [3].

A robust architecture is one that is flexible, changeable, simple yet elegant. If done right and documented well, it reduces the need for interteam communication and facilitates successful implementation of complex modules. If done well, it is practically invisible; if done poorly, it is a never-ending source of aggravation, cost, and needless complexity.

Architecture flows from the requirements and the functional specification. The requirements and functional specification need to be traced to the architecture and its modules, and the modules in the architecture should be traced to the requirements and functional specification. The requirements must necessarily be correct, complete, unambiguous, and, where applicable, measurable. Obtaining requirements with these qualities is the responsibility of the architect. It must be his highest priority. He does this by interacting closely with the customers and domain experts. If necessary, he builds prototypes to validate and clarify the requirements The architect acts as the translator between the customers and the developers. The customers do not know how to specify their needs in the unambiguous language that developers need, and the developers do not always have the skills to do requirements analysis.

The architect communicates his desires to the developers by specifying black-box descriptions of the modules. Black boxes are abstract entities that can be understood, and analyzed independently of the rest of the system. The process of building black-box models is called abstraction. Abstraction is used to simplify the design of a complex system by reducing the number of details that must be considered at the same time, thus reducing confusion and aiding clarity of understanding [4]. For safety-critical, military-critical, and other high-integrity sytems, black boxes can be specified unambiguously with mathematical logic using formal methods. Supplemented with natural language descriptions, this is probably the safest way to specify a system. It is usually more expensive and time consuming, as well. In the future, however, all software architects should know how to mathematically specify a module.

A robust architecture is necessary for a high-quality, dependable system. But it is not sufficient. A lot depends on how the developers implement modules handed to them by the architect.

The Rest of the Solution

Developers need to build systems that are dependable and free from faults. Since they are human, this is impossible. Instead they must strive to build systems that minimize faults by using best practices, and they must use modern tools that find faults during unit test and maintenance. They should also be familiar with the concepts of measuring reliability and how to build a dependable system. (A dependable system is one that is available, reliable, safe, confidential, has high integrity, and is maintainable [5].) In order for the system to be dependable, the subsystems and modules must be dependable.

"A key point [Hatton] stresses is to never incorporate stylistic information into the standard."

Fault prevention starts with clear, unambiguous requirements. The architect should provide these so the developer can concentrate on implementation. If the architecture is robust, the developer can concentrate on his particular module, free of extraneous details and concerns. The architect's module description tells the developer *what* to implement, but not *how* to implement it. The internals of the implementation are up to him. To ensure dependability, the developer needs to use sound software engineering principles and best practices, as these are his chief means of of minimizing complexity. Two best practices are coding standards and formal inspections.

Coding standards are necessary because every language has problem areas related to reliability and understandability. The best way to avoid the problem areas is to ban them, using an enforceable standard. Les Hatton describes why coding standards are important for safety and reliability and how to introduce a coding standard [6]. A key point he stresses is to *never* incorporate stylistic information into the standard. It will be a never-ending source of acrimony and debate. Such information, he says, should be placed in a *style guide.* Coding standards can be enforced with automatic tools that check the code, and by formal inspections. The benefits of formal inspections for defect prevention are well-known and well-documented. They are also invaluable for clarifying issues related to the software.

Developers need to measure their code to ensure its quality. This provides useful feedback to the developer on his coding practices, and it provides reassurance to the system's acquirers and users. Many static metrics can be used to assess the code. Among these are purity ratio, volume, functional density, and cyclomatic complexity. As a doctor uses a battery of tests to gauge a person's health, relying on more than one metric and covering all his bases, a developer using static analysis tools can do the same [7].

A good metric, for example, is cyclomatic complexity. A large value is a sign of complex code, which may be an indication of poor thought given to the design and implementation. It is also a sign that the code will be difficult to test and maintain.

Fault detection by proper unit testing is vitally important. To be done right, it requires the use of code coverage and path analysis tools. Unfortunately, this type of testing is usually overlooked. Many managers say they cannot afford them. Somehow, though, they can afford to fix the problems after the software has been fielded. This is penny-wise and pound-foolish. It is axiomatic that fixing software faults after the code has been deployed can be up to 100 times more expensive than finding and fixing the fault during development [8].

Besides path analysis and code coverage tools, automatic testing tools should be used. Human testers cannot hope to match the computer on indefatigability or thoroughness. In large systems, if testing is not automated, it is not done, or done rarely. For example, regression testing, used in systems undergoing modification and evolution, is essential to ensure that errors are not injected into code undergoing change, a very common problem in complex systems. Without automation, the process is onerous and time consuming. It rarely gets done, if at all.

Developing quality code is not simple or easy. It requires discipline and rigor, state-of-the-art tools, and enlightened managers willing to support developers by paying up-front costs, such as giving developers more time to automate and test their code. Developers take pride in their work. When they get the support they need, they know that their managers want them to produce quality code. This makes the work satisfying and rewarding.

Summary

Managing and limiting complexity and promoting clarity is fundamental to developing large software systems. The key ingredient is a robust architecture. The conceptual integrity of the architecture, its elegance and clarity, depends on a single mind. Developers build upon the architecture and ensure its robustness by rigorous application of basic software engineering principles and best practices in their code development.

References

1. Johnson, J., *Turning Chaos into Success,* www.softwaremag.com/archive/1999dec/Success.html, Dec. 1999. Standish Group.
2. Brooks, F. P., *The Mythical Man-Month: Essays on Software Engineering,* Anniversary Edition. Addison-Wesley, 1995.
3. Bass, L., P. Clements, and R. Kazman, Software Architecture in Practice. Addison-Wesley, 1998.
4. Berzis, V., and Luqi, *Software Engineering with Abstractions.* Addison Wesley, 1991.
5. Lyu, Michael R., Editor, *Handbook of Software Reliability Engineering.* IEEE Computer Society Press, 1995.
6. Hatton, L., Safer C, *Developing Software for High-Integrity and Safety-Critical Systems.* McGraw-Hill International Series in Software Engineering, McGraw-Hill International, 1995.
7. Drake, T. Measuring Software Quality: A Case Study, *IEEE Computer,* Nov. 1996.
8. Boehm, B. W., *Software Engineering Economics.* Prentice Hall, 1981.

About the Author

John Evans is a software engineer at SPAWAR Systems Center in San Diego (SSC-SD), where he has worked for the last 16 years. His job is to improve the software processes of projects within the Intelligence, Surveillance, and Reconnaissance Department (D70), and help the Software Engineering Process Office (SEPO) of SSC-SD improve the center's software maturity. He received his master's degree in software engineering from the Naval Postgraduate School in 1997 via distance learning, and SSC-SD sponsorship. He is now working on his doctorate in software engineering, under the same auspices.

SPAWARSYSCEN D73C
53570 Silvergate Ave., Room 1047
San Diego, Calif. 92152-5182
Voice: 619-553-5479
Fax: 619-553-5499
E-mail: evansjr@spawar.navy.mil

Common Sense–Can You Dig It?

Processes are a good thing. I am a PSP instructor, and appreciate how a good process can make my work habits more productive and increase personal quality. In fact, I was discussing with my lovely and charming wife about my idea to create a Personal Garden Planting Process (PGPP). She was quite amused with my PGPP, but pointed out that the process was quite useless.

"Never," I countered to this heresy. I had considered everything from soil moisture to length of the grass. She pointed out that there were several inches of snow in the yard—my process had neglected to consider the uselessness of planting a garden in the middle of winter. Good process—poor timing and implementation. Which brings us to Mr. Adams, Mr. Baker, and Mr. Charles

Mr. Adams, Mr. Baker and Mr. Charles were three software engineers in a particular Department of Defense agency who suddenly found themselves without a job. It seems the agency, after doing an A-76 study, contracted out the work and transferred all but these three software developers to other jobs.

Unfortunately, these men were within three months of retirement. Management, the big softies, arranged to have them perform other chores within the organization for three months, and then take retirement. Mr. Adams, Mr. Baker, and Mr. Charles were offered the job of site beautification. Their job was to plant shrubbery around the various buildings.

Being trained software engineers, the three men got together and came up with a process. Every evening, they ran the automatic sprinkler system to soften the ground. Early in the morning, Mr. Adams would start digging the holes for the shrubs. Mr. Baker would follow him, spreading fertilizer and inserting the shrubbery. Finally, Mr. Charles would cover the roots with dirt.

One day while parking his car, the big boss was puzzled to see Mr. Adams busy digging holes that Mr. Charles immediately filled with dirt. He expressed his amazement, only to be told that Mr. Baker was ill that day. But as Mr. Adams and Mr. Charles explained, "There's no reason to abandon our process just because one person isn't following it."

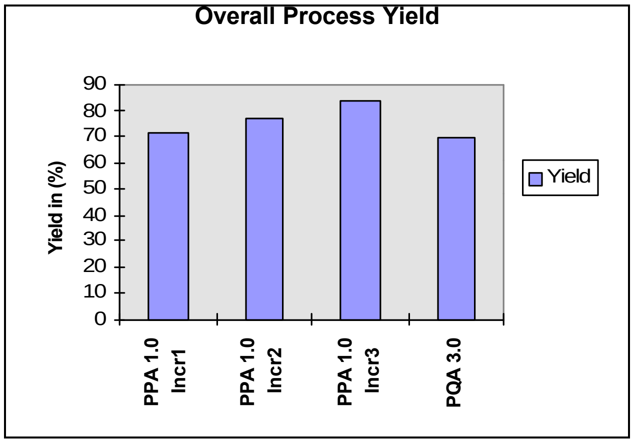
It is important to have a process—it serves as a road map. It lets you know how to get to where you are going. If you don't know where you are going, no matter how good the road map is, it doesn't help. It is important to understand the purpose of the road map, your goals, and the rationale of your process. Know when to follow the process—and know when the process will not work. When it does not work, it is time to modify the process. Processes are important. So is common sense.

—Dave Cook, Draper Laboratories Inc.

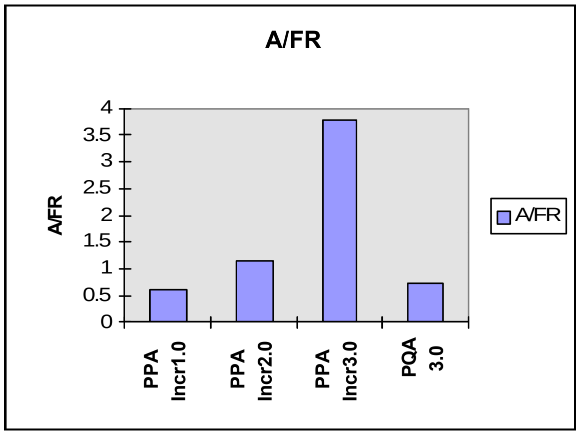
CALL FOR ARTICLES

software acquisition	Submit by July 1, 2000
project management	Submit by Aug. 1, 2000
modeling & simulation	Submit by Sep. 1, 2000
configuration management	Submit by Oct. 1, 2000
measures & metrics	Submit by Nov. 1, 2000
requirements management	Submit by Dec. 4, 2000

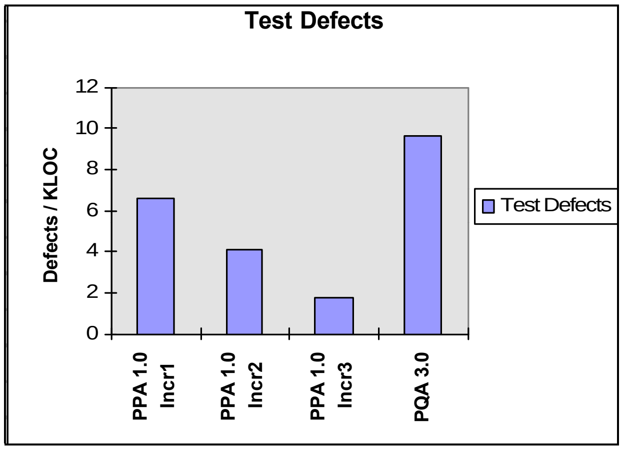
E-mail submissions to features@stsc1.hill.af.mil
Author Guidelines may be found at www.stsc.hill.af.mil



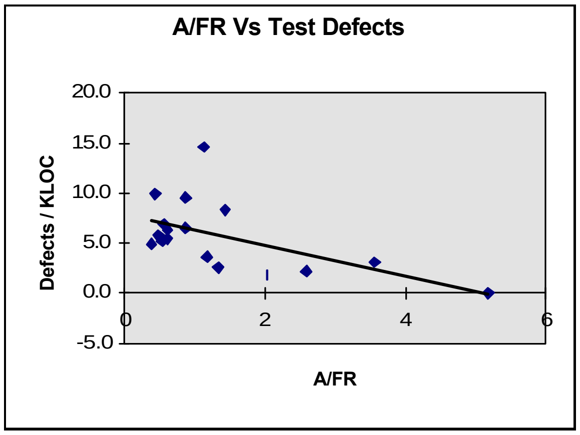
Higher Yields



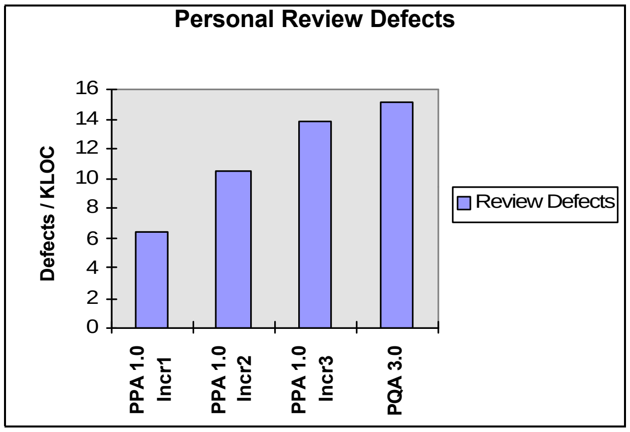
Greater A/FR



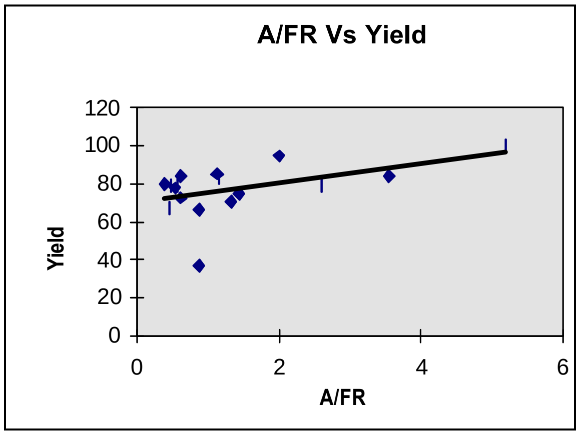
Fewer Test Defects



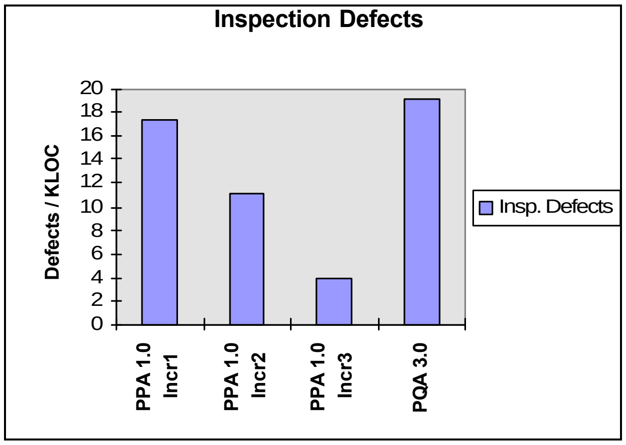
A/FR Vs. Test Defects



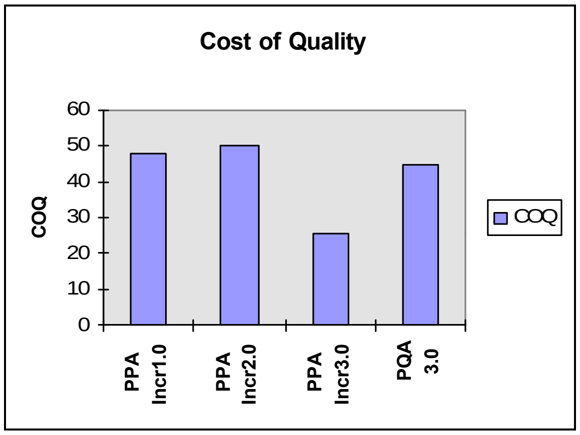
More Defects found in Personal Reviews



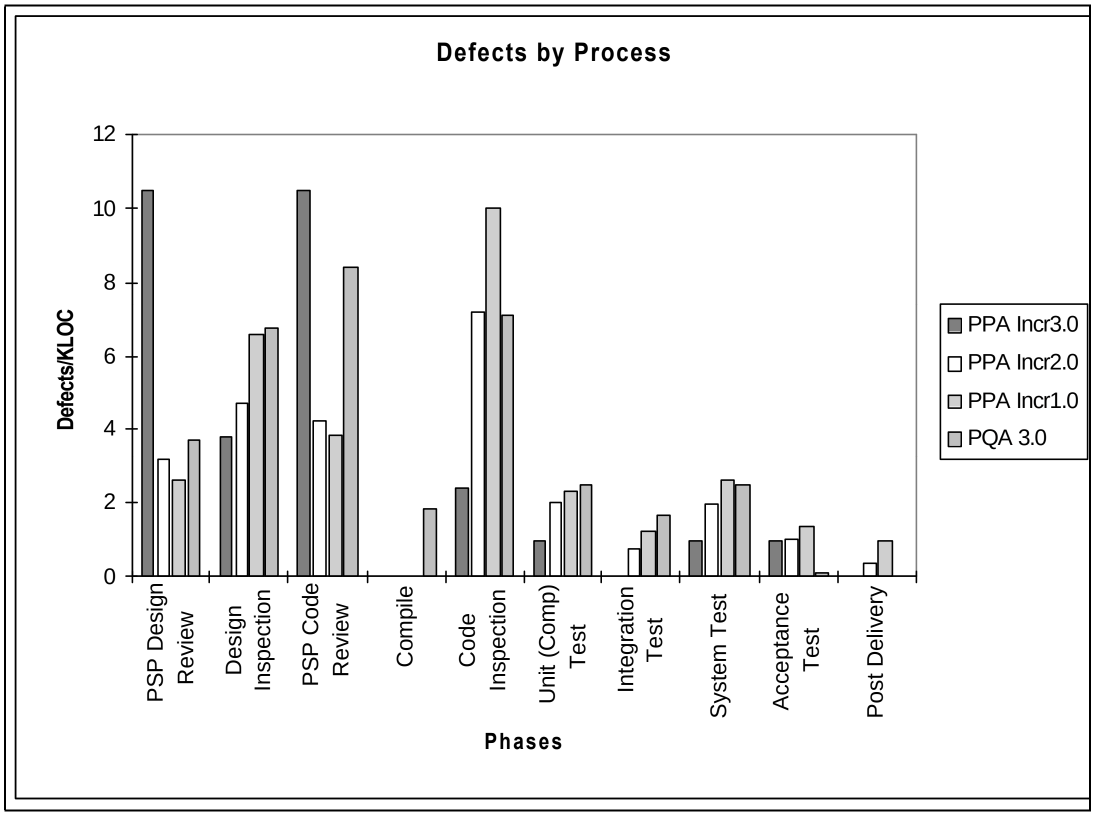
A/FR Vs. Yield



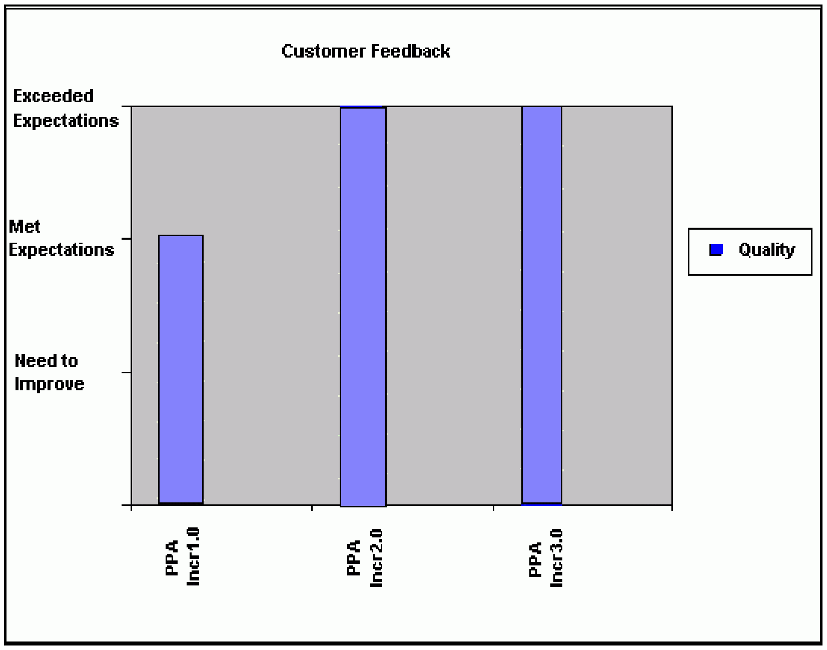
Fewer Defects found in Team Inspections



Cost of Quality Declined



Defects by Process



Customer Expectations Exceeded