**Information • Knowledge • Systems *Management*** 3 (2002/2003) 81–85
IOS Press


# Software Engineering Practices
## *Are Software Quality and Time to Market Incompatible?*
## An Interview with Steve Cross

Stephen E. Cross is the Director and Chief Executive Officer of the Software Engineering Institute (SEI), a United States Department of Defense sponsored Federally Funded Research and Development Center (FFRDC) situated as a college-level unit at Carnegie Mellon University. Dr. Cross is a member of the Air Force Scientific Advisory Board and the past chairman of the Defense Advanced Research Projects Agency (DARPA) Information Science and Technology (ISAT) panel. He is a Fellow of the Institute of Electrical and Electronic Engineers (IEEE). Dr. Cross received his Ph.D. from the University of Illinois, his M.S.E.E. from the Air Force Institute of Technology, and his B.S.E.E. from the University of Cincinnati. He retired from the U.S. Air Force in 1994. Dr. Cross was profiled in a Jan. 4, 2002, article titled "Gurus of Tomorrow's Technology" in the online edition of Business Week.

**IKSM:** Software is perceived to be an increasingly expensive portion of system development and deployment. Is this true? If so, why?

**Cross:** While we usually think of software as the stuff that allows our desktop computers and servers to "do their thing," the reality is that today software is the critical component in almost every system being brought to the marketplace. Cell phones, PDAs, medical equipment, entertainment devices, home appliances, and automobiles are all software-intensive systems. Software's cost, and development and deployment times, are increasing for three reasons: (1) there is more of it, (2) it is being used for more complex tasks, and (3) it is too often developed using inadequate and undisciplined methods.

For example, consider the significant increase in software-controlled devices in automobiles. Since the 1980's, automobiles have used electronic control units (ECUs) to perform system-monitoring tasks as well as to support diagnostics. Today's high-end automobiles have up to 70 ECUs and they are being used for new tasks, such as fuel management. The amount of software in Ford vehicles has increased by 16 times during the 1990's. For just one subsystem, engine management, estimates predict that the amount of software will increase by 50% to more than five million lines of code over the next five years. The industry average for developing a new powertrain control module is 10 software engineers and three years. Such software is more complex because of its size and because of its high demands for real-time performance, reliability, and safety. You can't reboot an automobile. In addition, the new field called telematics — put simply, the merger of telecommunications and computers in automobiles — is introducing new software-based services into automobiles. For example, telematic systems like OnStar will be standard equipment on 85% of the world's automobiles by 2010.

While software size, complexity, and functionality are increasing, the methods for building the software are not keeping pace. Software is too often developed using "test-in-quality" methods. By "test in quality," I mean that 60–80% of the time and cost of software development occurs after the coding phase. Defects are found during test, and software rework is required to repair the defects. This is akin to the situation found in other manufacturing fields 30 years ago. Manufacturers learned that it

was much more expensive to remove defects than it was to prevent them in the first place. Effective methods were instituted to reduce costs and schedule delays. Coding is the "manufacturing" of software. More attention must be paid to using better design methods that prevent defects from being introduced during coding.

**IKSM:** You've made the case for the increasing pervasiveness of software for accomplishing increasingly complex tasks. However, why are people using "inadequate and undisciplined methods" to develop this software? Is there something about software that is different from the "complex systems" of the 1960's, '70's and '80's — for example, aerospace and transportation systems?

**Cross:** First, developing software is just plan hard. By its very nature it is complex. The complexity is derived from the branching logic inherent in the code, the fact that software can be designed to adapt — to change functionality — and a whole host of other reasons. Chief among these is that software developers may focus on ensuring that the software does what it is supposed to do, but do not often think about what the software is not supposed to do. An example is a buffer overflow. Buffer overflows have been a problem in software-based systems and applications for a long time, and they are still the leading cause of cyber-security incidents. One of the first significant computer break-ins that took advantage of a buffer overflow was the Morris worm, and that happened in November 1988. The worm took advantage of a buffer overflow in the "finger" service that dispenses information about the set of users logged into a UNIX-based computer system. Even though the cause, and the necessary countermeasures, have been well publicized, buffer overflows are still a major cause of intrusions. A buffer overflow is possible because the creator of a computer program writes code that does not properly check the size of the destination area or buffer to see if it is big enough to completely hold its new contents. It's like trying to stuff the proverbial 10 pounds into the 5-pound bag. If the data intended for the new home doesn't fit and spills over, it can create a mess that requires somebody's attention, usually the computer owner's or a system administrator's.

Second, software becomes obsolete very quickly. In the book *Secrets of Software Success* the authors document numerous cases where commercial software became obsolete within 18 months. (Hoch, D., et al., *Secrets of Software Success: Management Insights from 100 Software Firms Around the World*, Harvard Business School Press, Boston, MA, 2000.) How many of your readers still use VisiCalc, the first commercial spreadsheet, or Word Perfect, the first commercial word processor? Both lost market dominance when they delayed product releases and their competitors did not. Commercial software products are continually reinvented. This reinvention means that product upgrades have more features — more functionality — and that makes the resulting products more complex. The focus is on getting the product into the marketplace faster than the competition does. This rush to market often means doing whatever it takes to get the software built, even if it means using less-than-perfect development practices and introducing products with defects.

**IKSM:** To what extent are these phenomena — product complexity and market pressures — unique to software? Microprocessors would seem to have similar characteristics. Do they encounter similar problems?

**Cross:** Let me give you two non-software examples. Guy Kawasaki has described Apple Computer's approach to the early Macintosh computer as "don't worry, be crappy." (Kawasaki, G., *Rules for Revolutionaries: The Capitalist Manifesto for Creating and Marketing New Products and Services*, HarperCollins Publishers, New York, NY, 2000.) Part of his message was that when you give the public what they want, you do not have to be terribly concerned with quality. That is, the product may not be as good as what you could have achieved, but what's important is to be the first

one to the marketplace with the item the public wants. A second example happened several years ago when a defect was discovered after Intel released a new microprocessor. Subsequently it was shown that formal method techniques could have been used to represent the logical design of the microprocessor and detect the defect. Those are two non-software examples, but they seem to be exceptions. With software, such stories are the expectation.

Detection of defects is very difficult with software. First, software is designed to achieve a set of requirements in a modeled world — it need not obey natural physical laws or even logic. This makes it much more difficult to automate defect detection and adds to the inherent complexity of software. Testing takes an inordinate amount of time and there is no guarantee that thorough testing will find all the defects. In fact, there was a famous bit of research done at Hitachi Software many years ago showing that the defects discovered during software test correlated with the defects yet to be discovered; a lot of defects found early correlated to defects that were yet to be found. (Yamaura, T., "How to Design Practical Test Cases," *IEEE Software*, November/December 1998, pp. 30–36.)

Another thing to consider is that software can be very quickly delivered to the market. Software requires no manufacturing capability and no large distribution systems. It can be downloaded over the Internet. This, and the idea I mentioned earlier about being first to market, are drivers in the marketplace. The consumers of software, who often do not have high expectations about quality, or do not know that high quality is possible and therefore do not demand it, also desire new features and functionality. Hence our "don't worry, be crappy" software culture.

**IKSM:** Does this attitude differ for mission-critical systems, weapon systems, safety systems, and similar "must work" systems? How much of this phenomenon is due to the nature of software and how much to the nature of typical applications?

**Cross:** Each system is built, consciously or subconsciously, as a reflection of the business goals that are critical within the organization developing the software and the organization in which the software is intended to be used. We think that product quality attributes — reliability, security, usability, modifiability, portability, and others — are a reflection of those business goals. In safety-critical and mission-critical systems, reliability is paramount. In a desktop application, usability and functionality are the principal attributes of interest. It is not possible to have 100% adherence to all desirable product quality attributes; one needs to seek a balance. Given a goal of reliability, or safety, the state-of-practice approach is to test extensively. For example, it is not uncommon for satellite software to undergo testing for years. Though time-consuming and costly, such extensive testing has resulted in software-intensive systems that are highly reliable (Humphrey, W., *Winning with Software*, Addison-Wesley, Boston, MA, 2002, pp. 51–53.).

**IKSM:** It seems, then, that we get exactly what we want, as demonstrated by what we are willing to pay for. This suggests that there is no "software problem." We don't, for example, like watching Windows' "blue screen of death," but we are demonstrably unwilling to pay for software without such problems. Where is this reasoning off?

**Cross:** Anyone who has experienced a system crash and has lost the file he or she was working on may disagree with you. But the real underlying issues are not very well understood. As we previously discussed, poorly designed, defect-riddled software is easily exploited by intruders, cyber terrorists, and the like. This is a growing problem. And there is an interesting economic model in play here in which the marketplace, acting as the beta tester, absorbs the costs of developing high-quality software. This software improves with each build, maybe over a period of years. If it were designed defect-free in the first place, would people be willing to pay a higher price? Would there be a need for a higher

price if the level of effort to do it right the first time were significantly decreased? These are interesting questions that we are just beginning to explore here at Carnegie Mellon, both through the work of the SEI and with campus colleagues who are economists.

**IKSM:** Let's consider the prospects for what you are suggesting. It has been argued that the high level of complexity of software precludes its complete testing. The assertion that no algorithm can verify that a formal structure does what it is intended to do is argued to be provably true. How does this assertion square with your aspirations for defect-free software?

**Cross:** In *Augustine's* Laws (Augustine, N. *Augustine's Laws*, AIAA Press, 6th Edition, 1997, pp. 75–80), the 12th law is the "First Law of Counterproductivity." Simply stated, "it costs a lot to build bad products." Norman Augustine describes a study by David Garvin at the Harvard Business School on the relative costs of air conditioners manufactured in the U.S. and Japan. The Japanese air conditioners cost substantially less because the rework due to product recalls was so much less. That is, the Japanese air conditioners were manufactured essentially defect-free with 100 times fewer defects than the American versions. The lesson learned here is that defects detected after manufacturing are more expensive to correct than those detected early.

Currently, software testing occurs after software development. Software development consists of the design activities required to specify what the software is supposed to do and the translation of those specifications into actual code. Formal method approaches enable one to prove that the code is implemented correctly, where correctness means that the code correctly implements the specifications. But there are other kinds of defects, such as those related to errors introduced during design. For example, the specifications are not right: they do not reflect what the code is supposed to do, but rather what the designer assumes the end user wants to do, or there are unintentional consequences of the design requirements that are not considered.

When these kinds of defects are detected after coding and during test, the cost of redesign — that is, rework — is high. What I'm advocating is the use of better engineering practices that put more emphasis on preventing the introduction of defects during design, because the best way to improve productivity and to lower costs is to prevent defects during manufacturing and during design. History seems to be repeating itself. Air conditioner development consists of design and manufacturing. Software development consists of design and coding. Coding is the manufacturing of software. Anything that can be done to detect or prevent defects during design results in greater productivity and decreased cost — whether it is software or any other engineered artifact.

**IKSM:** This makes sense if the designers and manufacturers of products have to bear the costs of defects. This does not seem, however, to be the case for much of software. End users often bear these costs. The upside for software developers, as you indicated earlier, is that they get their products to market faster. And the downside is — well, there does not seem to be a downside for the software vendors. How might this "equation" be changed?

**Cross:** I suggest the consumer is causing a shift right before our eyes. Let me give a few examples of how this is happening. In a recent issue of *Business Week*, Microsoft CEO Steve Ballmer indicated that the company's primary focus is now software quality (Greene, J., et al., "Ballmer's Microsoft," *Business Week*, June 17, 2002, p. 66–74). This followed Bill Gates's announcement of Microsoft's trustworthy computing initiative, which was motivated in part by cyber security concerns. However, as described in the Ballmer interview, at his first off-site, frontline Microsoft executives expressed grave concerns about their market segment's negative perceptions of Microsoft's commitment to quality.

This is one example of how the consumer and the marketplace are causing a shift, albeit slow, in the current culture, in which "first to market" is all important.

Consumers are also taking proactive action. In a recent issue of *CSO Magazine* there is a description of how contracts between consumers and vendors increasingly contain language — often called code integrity warranties — requiring software quality. (Berinato, S., "The Big Fix," *CSO Magazine*, October 2002, pp. 30–36, available online: http://www.csoonline.com/read/100702/fix.html.). As the consumer community shares this language and the legal community refines and adopts it, there will be another tangible way to give vendors incentives to address quality issues. Slowly but surely, quality will be viewed as an equal differentiator with speed to market. The irony here is that the two are not mutually exclusive. In fact, the use of disciplined engineering techniques actually leads to development acceleration.

Besides marketplace expectations brought to bear by consumers, there are other forces in play that will cause a change in the culture and current paradigm. Insurance companies such as AIG now write cyber security insurance to protect companies that are increasingly reliant on the vulnerable information infrastructure. There are precedents in how the insurance industry has been the catalyst for improving the engineering and safety of systems. Perhaps the best example is from the 19th century with steam boilers. As steam power was used more and more to spawn the industrial revolution, accidents from poorly engineered and manufactured boilers became a real problem. Standards of engineering excellence were imposed on boiler manufacturers and their consumers — factories, railroads, and others — as part of insurance contracts. The marketplace, as motivated by security concerns and the insurance industry, will help migrate the change to a speed and quality paradigm.

These are three examples I see playing out in real time. Let me just add that my personal hope is that the private sector, led by the marketplace, will guide the change to embracing software quality. I'd hate to see the government get involved in legislating or otherwise regulating software quality.

**IKSM:** Your arguments are compelling. If an enterprise buys into your line of reasoning, what should it do to pursue this change? Does your guidance differ for private and public sector enterprises?

**Cross:** They are free to contact me directly. If the enterprise is a developer of software, there are courses at the SEI and "how to" information on our Web site, www.sei.cmu.edu. There are also numerous other sources of helpful information through professional societies such as IEEE and ACM, and other universities and companies. If the enterprise is part of company that is a software consumer, its contract staff should be getting smart on new contract language, like code integrity warranties, and writing them into their contracts. And every consumer should start asking the vendors of their software if the code is developed with disciplined software engineering practices and how many patches they released in the past year that were attributed to software defects.

**IKSM:** Thank you, Dr. Cross, for an illuminating view of emerging changes in software development practices. Purchasers, consumers, and users of software, without doubt, will look forward to the changes you portray.