

Focusing Software Education on Engineering

John C. Knight
Department of Computer Science
University of Virginia

“We must decide we want to be engineers not blacksmiths.”
Peter Amey, Praxis Critical Systems

Introduction

The software crisis is still with us. In fact, it is worse than it has ever been, and we see evidence of the crisis regularly. All manner of applications from desktop systems to large-scale information systems are delivered late, exceed their projected budgets, and fail in various ways leading to inconvenience, loss of service, and loss of revenue. A recent study by the National Institute of Standards and Technology found that software errors cost the U.S. Economy about \$59.5 billion annually [4].

All aspects of the crisis are important including the losses, but there are applications, usually referred to as *safety-critical* applications, where failure has very significant consequences. Developers of the software for such systems are expected to do the utmost to prevent failure, and in many domains government regulation prescribes things like development processes, test metrics, and so on. Despite the goals, many safety-critical systems fail, often with spectacular consequences, and investigations of such failures often document software defects as contributory causes.

Failures of safety-critical systems where software is a contributory cause occur for many reasons, and the obvious question is: “What should be done about the situation?” All sorts of explanations have been offered and all sorts of technical solutions proposed. Obviously there are many things that could be and are being done, but for the most part ongoing work is looking for technical approaches. In an earlier paper [1], I called for attention to be paid to the basic education that software engineers receive. I did this for two reasons:

- Many of the problems in safety-critical software arise from elementary mistakes. They could have been avoided easily if the software engineers involved were properly educated.
- Most software engineers get all the formal education they will ever receive in undergraduate degree programs. Unless these programs educate engineers in an adequate way, the situation is unlikely to change.

In this paper I raise a different but related issue, specifically that engineers educated in non-software fields often write software, including software for safety-critical systems, but they should not do so unless great care is taken. Engineers in disciplines outside of software engineering frequently fail to realize how difficult the development of safety-critical software is. This failure leads to non-software engineers undertaking software development even though their education does not qualified them to do so. This situation arises frequently in my experience, often with

serious consequences. To solve it, we need to inform the traditional engineering disciplines of what we know about the capabilities and limitations of software engineering. In other words we need to focus *software education on engineering*.

Failures Of Modern Safety-Critical Software

The spectrum of system failures and the associated range of technical issues that need to be faced by the traditional engineering community is very broad. There have been numerous serious operational failures of safety-critical systems in which software was identified as one of the contributory factors. Examples include the Ariane V launch vehicle failure, the crash of Korean Air Flight 801, and the loss of NASA spacecraft including the Mars Global Explorer, and the Mars Polar Lander.

Operational failures are not the whole story. Expensive difficulties often arise during development because of the need to ensure that software meets high dependability requirements. The Lockheed F22 Raptor fighter aircraft, for example, has been plagued with problems in software development for much of its development history. The FAA's Wide Area Augmentation System, a system designed to supplement the Global Positioning System, is now years late in delivery and way over budget. Perhaps the worst situation is one in which significant resources are expended on a system that never makes it into production. NASA's Checkout and Launch Control System for the Space Shuttle is an example. It was cancelled in September of 2002 after five years of development at a cost of roughly \$300 million. One has to regard huge losses like this as significant and view the underlying system as being "safety critical" in a sense because of the extreme financial consequences of failure.

Operational failures and development difficulties are not the whole story either. All might appear to be well during development and operation until a defect is detected in a support tool that was used. Support tools are not regarded typically as safety-critical systems although to the extent that they contribute to the development or operation of a safety-critical system, they have to be. As an example of this issue, consider the following. On May 20, 1996, the Nuclear Regulatory Commission issued a notice to "All holders of operating licenses or construction permits for power reactors", the purpose of which was to inform addressees of defects that had been reported in structural analysis software. This software might have been used to analyze reactor pressure vessels although the Commission did not know which plants had used the software and which had not. Several software defects had been reported, and they might well have affected the analytic results upon which safety arguments rested.

Finally, I note that security of information systems is a serious and growing problem as the world becomes interconnected and more of our critical functionality is subsumed by software-intensive systems. As such, the security properties of a system become crucial and the losses from failures can be extraordinary. Apart from the obvious possibility of funds actually being stolen, there are the losses that arise from worm attacks that disable systems, from denial-of-service attacks, and from violations of information integrity and privacy. Security vulnerabilities are, for the most part, attributable to software defects, and it is clearly the case that many high-value systems can and do suffer extensive losses from software defects that masquerade as security problems.

In practice, operational failures are usually the most visible because of the immediate loss, but developmental and security failures are costly also. The challenges faced by developers building safety-critical systems are many, and dealing with them properly requires many skills. But this raises the question of whether technology can solve or mitigate the problem.

Applying Software Technology

A great deal is known about how to build software, and in many cases (but far from all cases) the technology is taught well. In principle, complex software systems can be built. So what goes wrong when we try to build significant software systems? Of course, there are many things that go wrong but in this section I discuss two examples.

A major factor that underlies a lot of defects is requirements change. We know that the requirements for a system are difficult to capture and a lot of effort has been expended on development of techniques in the field of requirements engineering. Many of the benefits that accrue from careful requirements engineering are lost, however, because engineers building the remainder of the system are unaware of the critical importance of requirements accuracy. And they assume that software is easy to change if the requirements change. This is not the case, in general, and leads to fatally flawed system development processes.

Implementation, the process of creating high-level language programs from specification, is generally viewed as a process that requires human creativity. Using appropriate techniques of both synthesis and analysis, properly educated software engineers can develop high-level language programs that are remarkably free of defects. Yet creation of high-level language programs remains a complex undertaking and defect rates can be surprisingly high. In a study undertaken by Yu [2], a wide range of defects in large numbers were found in production telecommunications software. Defects seen included the use of uninitialized variables, misuse of break and continue statements, incorrect order of operand evaluation because of misunderstandings of operator precedence, incorrect loop boundaries, indexing outside arrays, truncation of values, misuse of pointers, and incorrect AND and OR tests. That this was the result of imperfect engineering is indicated by the fact that simple techniques were developed by Yu to help reduce the number of defects and the result was a reduction of approximately 34% between one system release and the next. The cost of the project was quite small and was dwarfed by the cost savings.

Modern software technology is quite good but it is applied poorly in many cases with the result that software costs more than it should and contains more defects than it should. So what needs to be done about this situation? Would it be productive to continue to seek technological solutions to the problem? Would better tools, programming languages, processes, test techniques, or metrics be a way to make the state of software better? Certainly there are directions that can be taken but they will not deal with the fundamental problem. In addition to everything else that we do, we need to focus *software education on engineering*.

Focusing Software Education on Engineering

The traditional engineering disciplines are not software engineering. Modern engineered systems are software intensive. Software is involved in their design, manufacturing, operation, and main-

tenance and all traditional engineering disciplines must appreciate this. The elements of software engineering that arise in developing a modern system might include formal specification, advanced design techniques, operational concurrency, real-time operation, memory management at all levels of the hierarchy, process and development risk management, verification, dependability assessment and assurance, and process resource management. In many ways, each of these and similar topics are sub-fields, and each has a substantial body of knowledge. A software engineer is expected either to know the associated body of knowledge and to practice it or to be aware that they do not know it and to seek help. Knowing that you are not qualified to undertake some element of software development is acceptable and just as important as knowing something. Engineers in traditional disciplines are unlikely even to be aware of what technical elements they need to know in the software field.

Most traditional engineering disciplines are associated with a professional organization that helps to maintain professional standards. In many cases, such standards are supported by codes of ethics with which professionals are expected to comply. Interestingly, in many cases, such codes include an explicit statement about practice based on one's abilities. As an example, the following text appears in the fundamental canons of the code of ethics of the American Society of Mechanical Engineers (ASME) [3]:

1. Engineers shall hold paramount the safety, health and welfare of the public in the performance of their professional duties.
2. Engineers shall perform services only in the areas of *their competence*. (my emphasis)

Given the importance of safety-critical systems and the complexity of the technology involved in software development, an ASME member who is not educated appropriately in the relevant software techniques but who is developing software for any system with significant consequences of failure would appear to be in violation of this ethical statement.

What Should Be Done?

The software engineering community needs to ensure that engineers at all levels in traditional engineering fields have a comprehensive understanding of the issues surrounding software development and their own limitations in that area. By all levels, I mean engineers engaged in formal education at a university all the way up to senior managers making engineering decisions. This understanding needs to cover four major topics each of which might require extensive educational effort. The four areas are: (1) the role of software in engineered systems; (2) the state of software engineering technology; (3) the limitations of current software engineering technology; and (4) the responsibilities of every engineer when dealing with software as an entity and with software engineers as professionals.

How might this level of understanding be achieved? It is crucial that appropriate "software literacy" courses be developed for traditional engineering fields. Such courses need to become mandatory in degree programs and need to emphasize the critical fact that software is pervasive in engineered systems. It is also time to consider government regulation, insurance requirements,

and monitoring by professional societies as approaches to enforcement of the necessary care in software development.

And Finally

The problem identified in this paper is really part of a wider issue. Both this problem and the lack of adequate education that even software engineers receive are symptoms of the lack of a comprehensive software engineering culture. Software is a crucial component of a growing fraction of the engineered systems that we build. It is often the pacing item in system development and is usually a significant fraction of the development cost. The rate of failure and the escalating cost of software are unlikely to be brought under control without an appropriate engineering culture in the software field.

References

- [1] Knight, John C., Should Software Engineers Be Licensed? Safety-Critical Systems Club Newsletter, Volume 14, Number 1, September 2004.
- [2] “Yu, Weider, A Software Fault Prevention Approach in Coding and Root Cause Analysis, Bell Labs Technical Journal, Volume 3, Number 2, April-June 1998, pp. 3-21.
- [3] American Society of Mechanical Engineers, Professional Code of Ethics, http://www.asme.org/asmae/policies/pdf/p15_7.pdf
- [4] National Institute of Standards and Technology, The Economic Impacts of Inadequate Infrastructure for Software Testing, Planning report 02-03, May 2002, http://www.nist.gov/public_affairs/releases/n02-10.htm