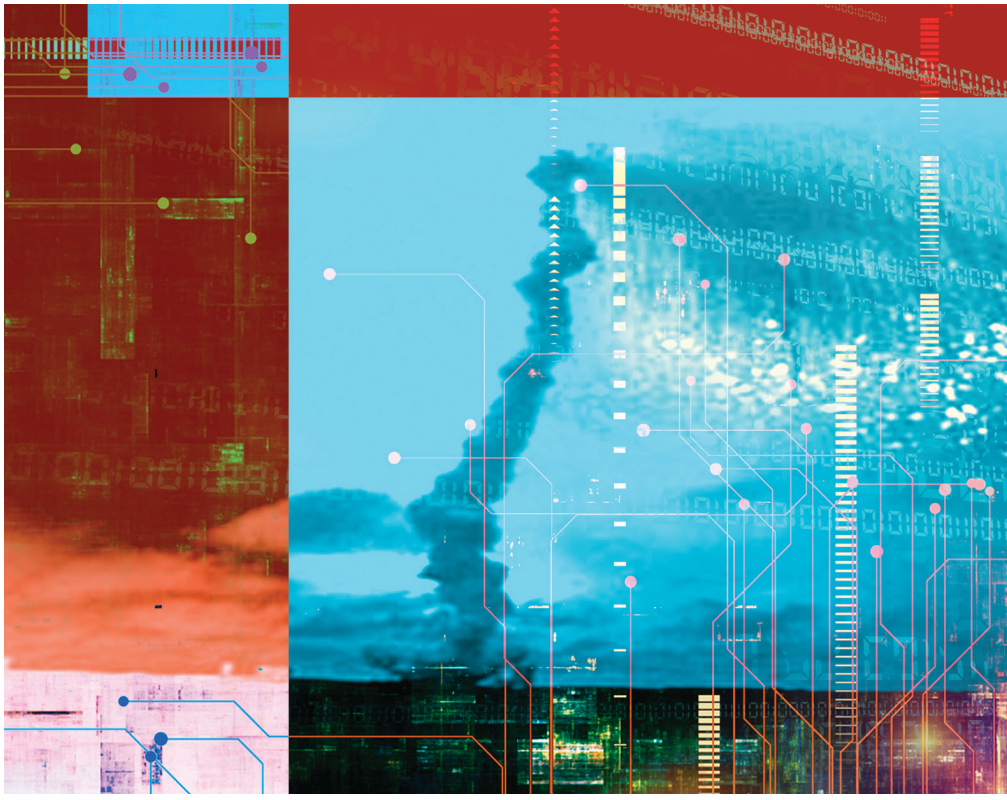
DOI:10.1145/2076450.2076463

Marvin V. Zelkowitz

Viewpoint
What Have We Learned About Software Engineering?

Upon closer examination, everything old appears to be new again in the realm of software engineering.



In late 2010, a *New York Times* headline attracted my attention: "A Pinpoint Beam Strays Invisibly, Harming Instead of Healing—A Radiation Setting Is Wrong, and Patients are Harmed."[a] I did not immediately learn the cause of the *New York Times*-reported incident, particularly if the cause was software-related, but it sure seemed a lot like the Therac-25 story of the mid-1980s.[2] The Therac-25 was an earlier medical device involved in several accidents where some patients were given fatal instead of therapeutic doses of radiation. I have since learned the problem reported in the *Times* involved passing information among three incompatible computers.[3] We apparently never learn.

The real message of the Therac-25 incidents was not that there was a software bug, but that software engineers missed a key engineering principle in designing that device. Any competent designer should be able to build software that detects a failure and either corrects it or responds in a safe manner. The problem with the Therac-25 was that a single error was compounded with a second error, and the device was not designed to handle multiple points of failure. Hardware engineers know how to build using multiple failure modes, something that was new to most software designers.

Software failures are well documented in the literature. On June 4, 1996 on its maiden flight, an Ariane 5 rocket exploded 38 seconds after launch.[5] Again, software was the cause. In this case, reusing unmodified software when the specifications for it changed and eliminating sufficient tests since "the code was correct" from the earlier Ariane 4 rocket were part of the problem.

Lesson Learned and Unlearned

The messages learned from such examples as these are critical for producing quality software. Software is a critical component of just about every device sold today. Even less safety-critical software has problems. The computer I am using to write this column

a *New York Times* (Dec. 28, 2010), A1; http://www.nytimes.com/2010/12/29/health/29radiation.html

ILLUSTRATION BY ANDRIJ BORYS ASSOCIATES

downloads a new "critical update" to some piece of software on my machine almost every day. Even my Windows 7 operating system seems to fail almost daily. What have we learned about producing good software? My general impression seems to be "Not much." As for the accident written up in the *New York Times* I mentioned at the beginning of this column, I could find no reference anywhere to the similarities to the Therac-25 accidents. Since the Therac-25 accidents occurred nearly 30 years ago, I assume it was well before most current professionals (both journalists and IT personnel) were plying their trade and the incidents are rapidly moving into the realm of ancient history.

So how are we in the U.S. responding to these problems? As described

by the U.S. Bureau of Labor Statistics, the largest growth in the computer field will "all require substantial training beyond the basic skills of an operator but not the scientific education of a computer hardware engineer. It isn't necessary to have a Bachelor of Science degree to be considered a software engineer."[1] My interpretation of this statement—a continual "dumbing down" of the ability of most software engineers is in store for the future.

I find it very frustrating when we are still talking about the same debugging techniques that were "old" when I started teaching in 1971.

For years Dave Parnas has been at the forefront in trying to get the field to regard software engineering as an *engineering* discipline, in deed as well as in name, by emphasizing good engineering principles in the curriculum of a computer science or related program.[7] However, even if successful, it makes little difference if most of the next generation of software engineers does not even have a Bachelor of Science degree.

What are we teaching the next generation of software engineers? I have always used the lessons of the Therac-25 and Ariane 5 as important concepts in system design. Testing, debugging, verification, and coding programs are important tools in any software engineering toolbox. But what are programmers actually using? Two examples: "..., there are people who find debuggers to be an inferior tool and who prefer to use in-program logging, or `printf`, statements to find out where their program is going wrong,"[4] and "Investing in a large amount of software testing can be difficult to justify, particularly for a startup company."[6] Those are concepts whose negative impact was well understood and taught about in the 1970s. Haven't we learned anything since then?

What has saved the software engineer is Moore's Law. For over 50 years computers have been doubling in speed about every two years, while at the same time getting cheaper. This has allowed inefficient and poorly designed programs to survive. But the era of ever cheaper and faster machines is rapidly ending. Heat generation and power usage have radically slowed down the production of ever faster processors since around 2005, and programming is becoming more difficult, not easier, in order to use new multicore processors effectively.[8] What will be needed for many applications are not under-qualified computer technicians, but better-qualified software engineers who understand the implications of parallel processing in addition to all the other technologies that have arisen in the quest for effective trustworthy software.

Conclusion

As a programmer since 1962 and a professor of computer science since 1971 I have tried to instill the ideals of the field in my students. But I find it very frustrating when we are still talking about the same debugging techniques that were "old" when I started teaching in 1971. It would be like in astronomy where each new generation of Ph.D.'s would have to first learn how to grind their own lenses as Galileo did 400 years ago before beginning their studies. How could physics and astronomy have progressed as much as they have if they were similarly restricted? Yet, we seem to be stuck reinventing the 1970s. I would hope we can do better. C

References

1. Grier, D.A. The migration to the middle. *IEEE Computer 44*, 1 (Jan. 2011), 1214.
2. Leveson, N.G. and Turner, C.S. An investigation of the Therac-25 accidents. *IEEE Computer 26*, 7 (July 1993), 18–41.
3. Neumann P.G. and contributors. Risks to the public. *ACM Software Engineering Notes 36*, 2 (Mar. 2011), 19–27.
4. Neville-Neil, G.V. Literate coding. *Commun. ACM 53*, 12 (Dec. 2010), 37–38.
5. Nuseibeh, B. Ariane 5: Who dunnit? *IEEE Software 14*, 3 (May 1997), 15–16.
6. Ortega, R. How much software testing is enough? *Commun. ACM 53*, 9 (Sept. 2010), 9.
7. Parnas, D.L. Risks of undisciplined development. *Commun. ACM 53*, 10 (Oct. 2010), 25–27.
8. Sutter, H. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal 30*, 3 (Mar. 2005).

Marvin V. Zelkowitz (mvz@cs.umd.edu) is a professor emeritus at the University of Maryland and a senior research fellow at the Fraunhofer Center for Experimental Software Engineering, College Park, MD.