Education of Software Engineers[1]

Marvin V. Zelkowitz

Professor Emeritus, University of Maryland, College Park, MD

Senior Research Fellow, Fraunhofer Center for Experimental Software Engineering,

College Park, MD

Introduction

As I was reading the *New York Times* at the end of 2010, the headline of a news article suddenly hit me - “A Pinpoint Beam Strays Invisibly, Harming Instead of Healing – A Radiation Setting Is Wrong, and Patients are Harmed” while undergoing SRS (stereotactic radiosurgery) treatment in a hospital. As American baseball player Yogi Berra once said “It’s déjà vu all over again.” When the story first appeared, it was not clear if the cause was software-related, but it sure read a lot like the Therac-25 disaster of the mid-1980s [1]. The Therac-25 was an earlier medical device where some patients were given fatal instead of therapeutic doses of radiation. A short time later I did read that the problem was in the programming of the SRS machine and involved passing information among 3 incompatible computers [2]. We apparently never learn.

In the case of the Therac-25, the problem was that the erase character key was not handled correctly, so if the code to switch between radiation and x-ray treatment was typed incorrectly and the backspace key was depressed, the machine would go into the wrong state. However, the real message of the Therac-25 was not that there was a software bug. Those happen all the time in programs and are generally fixed. However, in this case the software engineers designing the Therac-25 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. Fault detection and correction is standard fare for a competent software tester. The problem with the Therac-25 was that a single error was compounded with a second error. That is, the error in switching between radiation and x-ray modes was compounded by the error in the backspace key. The device was not designed to handle multiple points of failure.

[1] An earlier draft of parts of this paper appeared as Zelkowitz M. V., Viewpoint - What have we learned about software engineering?, *Comm. of the ACM* (55)2, 2012, 38-39. (Copyright owned by author.)

Hardware engineers know how to build products having multiple failure modes, but this is something that is still 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 blew up 38 seconds after launch [3]. In this case, a software register overran its limit and caused an incorrect value, interpreted by the control software as the rocket being out of control. Control was switched to the backup computer, running the same software. Since it too overran its limit because of the same software problem, the control program could only interpret that the problem was indeed true and had the rocket blow up.

Again, software was the cause, but that was not the real lesson from this experience. The designers of the Ariane 5 thought they were being smart by using the same code, which worked correctly on the older Ariane 4 rocket. However, the environment in which the Ariane 4 code was executing was different from that of the Ariane 5, and reusing the same code when its specifications changed without a thorough study of those specifications is a prescription for disaster. In this case the problem was compounded by eliminating sufficient testing since performance of the computer was an issue and "the code was correct" from the earlier Ariane 4 rocket. Such shortcuts are never a good idea. As a result of the failure, we again have that well-worn truism – "You don't have the money to do it right, but you do have the money to do it again."

I don't mean to just pick on the European Space Agency. NASA is also guilty of similar fiascos. Almost everyone knows of the Mars Climate Orbiter that crashed into Mars on November 10, 1999 due to a mix-up between Imperial measurement units and metric measurement units between two different components of software. A simple dimensional analysis between parameters and arguments in a subroutine would have revealed the problem.

Future Software Engineering Needs

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 non-safety-critical software has problems. The PC I am using to write this paper downloads a new "critical update" to some piece of software on my machine almost daily. Every day when I turn on my smartphone, two or three applications on the phone download a revised version of itself – usually to fix a problem. Applications on my Windows 7 operating system seem to fail regularly. What have we learned about producing good software? My general impression seems to be "Not much." As for the SRS accident I mentioned at the beginning of this paper, I could find no reference anywhere in the *New York Times* article to any similarities to the Therac-25 incidents of the early 1980s. Since that happened almost 30 years ago, I assume it was well before most current professionals (both journalists and computer personnel) were plying their trade and the incident is rapidly moving into the realm of ancient history. I have been a computer science researcher since the late 1960s, and there are few of us still around compared to the multitudes of programmers working today.

So how are we in the USA responding to these problems? I assume much of what goes on in the USA is similar in other countries. As described by the US 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*." [Emphasis added.] [4] I can't imagine any agency saying, for example, "Electronics are easy to fix today since you simply replace a bad component with a new one, so it is no longer necessary for electrical engineers to have college degrees." For some reason we haven't been able to get the message across to non-computer people that building software is actually hard.

My interpretation of the Bureau of Labor Statistics statement is that a continual "dumbing down" of the ability of most software engineers is in store for the future. While there are certainly talented professionals in the field, new employees will not be as competent as the present generation of engineers. Management seems to be looking for inexpensive solutions to solve the personnel problem rather than what is needed to eliminate the problem itself.

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 [5]. 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? When I taught classes in software engineering, I have always used the lessons of the Therac-25, the Ariane 5, the Mars Climate Orbiter, as well as other examples, 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? "…, 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" [6]. With all of the research in debugging systems and new testing tools filling conference proceedings, is anyone actually using that technology? As another quote, consider the following: "Investing in a large amount of software testing can be difficult to justify, particularly for a startup company." [7] What does this statement mean? A startup company is allowed to sell inferior code because testing a program is expensive? Would a new aircraft or a new automobile manufacturer ever make a statement like that? Are we allowing such products to be regularly sold? If software engineering were a true engineering discipline, those comments would be grounds for removing the licenses of all those involved in developing and selling such products.

Those are concepts whose negative impact was well understood and taught about in the 1970s. Haven't we learned anything since then?

Technology Saves the Day (So far)

Moore's Law

What has saved the software engineer up until now is "Moore's Law." For over 50 years computers have been doubling in speed about every two years, while at the same time becoming cheaper to purchase. However, the general public, and most computer experts, do not really understand what Gordon Moore really said in 1965. He claimed component density on a chip would double every 18 months with each component taking less space. At first this size shrinkage roughly translated into increased speed. But faster circuit speeds also meant more heat being produced, meaning more energy needed to power the chip.

For 40 years this has allowed inefficient and poorly designed programs to survive. But the days 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. To limit heat and power, most processors are no longer produced at clock speeds greater than 3 GHz. With users increasingly using smartphones and tablet computers, lightweight machines with long battery life is the driving force. The goal is to make machines smaller and faster using less energy per cycle. By limiting computation speed, but increasing the number of cores in each processor we continue to get the benefits of "Moore's Law" without generating increased heat. Thus the number of separate processors on each CPU chip now doubles according to "Moore's Law" every two years or so. Essentially each core is a separate cpu and 4 and 8 core processors are becoming common.

However, multiple cores make programming harder, not easier, in order to use these multicore processors effectively [8]. The complex algorithms developed over the last few decades that were part of the High Performance Computing (HPC) community's need to achieve petascale performance (i.e., 10^{15} floating point operations per second) must now be applied to the ordinary desktop machine. Although the top ranking of the currently fastest machines is now 17.59 petaflops[2], this speed was achieved on a highly specialized set of benchmark programs. While such petascale machines have been produced, in reality they run at only 10% or so of peak performance most of the time since programming such parallelism is extremely difficult.

[2] http://www.top500.org, November 2012.

What will be needed for many future 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.

Massive Open Online Courses

There are now two competing forces acting upon the university that has the possibility to greatly alter the landscape. On one hand, society is pushing back on the high costs of running a university. Universities, especially in the United States, have to do more with less money, so the need for increased revenue other than from student tuition or from the state is great. On the other hand, the World Wide Web has become the great equalizer and is location independent. If you visit a website, it really doesn't matter if it is in the same building as you, across the street, or across the world. From the user's perspective, they are all the same.

Universities have embraced the web with the massive open online course (MOOC). Once a course is broadcast on the web, it really doesn't add to the cost of running the course to have more students access the course website. Initially, some universities let non-paying students audit classes that local university students were taking for credit. In one early case, Canadian school Athabasca University offered a course to 25 tuition paying students and free to 2,300 others. However, the floodgates opened when Stanford University offered a course on artificial intelligence in the fall of 2011 to over 160,000 online students. Most were just auditing, but the issue that became apparent was how does a university commercialize such courses? If students want to take the course for credit, having them pay for it will solve most of the financial part. But how does the university handle the learning and evaluation part of any course? The number of teaching assistants needed to address 100,000 students in one course would overwhelm any university. (Assuming an optimistically high number of 1000 student enrollees per teaching assistant, a single MOOC would require over 100 graduate students for this single course. This permits only 20 minutes per student per course, a very low number if course grading and some advising is included in this total.)

This is where software engineering gets involved in this discussion. For some classes, automated tests can work. The technology to automatically grade short answer and multiple choice tests is available. But what about a course in software engineering? Software engineering is a group engineering laboratory activity. It requires group interactions and group decision making. What experience can a student get from working alone in a home office on a computer connected to the web? Do we know what can be done in order to make the MOOC concept work?

Evolution of Computer Technology

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 where 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 PhDs 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 did if so restricted? Yet we seem to still be stuck reinventing the 1970s. I would hope that we can do better.

One of the causes of this "reinvention of the wheel" in software engineering is that each generation of computer technology has been the product of a new generation of developers. In the 1960s and 1970s we saw the development of the large scale mainframes with large multiuser operating systems such as IBM's OS/360. System crashes were quite high at first, but eventually we learned how to make such systems reliable. UNIX, building on the research of the 1960s was initially released in 1970 and has been highly reliable [9].

While computer professionals were working on large mainframes perfecting complex code, the next generation of machines in the 1970s was being developed initially by computer hobbyists. The Apple II by Steve Jobs and Steve Wozniak in the mid-1970s and the MS-DOS software for the IBM PC by Microsoft founder Bill Gates in the early 1980s were essentially new developments, not built upon the foundation we learned about in the 1960s.

This trend continued through multiple generations of machines. The Android operating system developed by Google for the smartphone was generated by yet another generation of developers. While Android is a powerful operating system, applications seem to be updated regularly to fix errors. Only Apple seems to have done it right. In 2002 they released the forerunner of their latest operating system, OS X. This is based upon a UNIX kernel, which already was a thoroughly solid foundation in which to build a system.

Looking at all of these new generations of software and hardware and of the various failures described earlier, the solution cannot be less knowledgeable software engineers, but higher standards necessary for all.

A Strategy for Software Design

In order to address the problem of bad software, let me reinvent and describe a technique common in the 1970s, which seems to have been lost in today's development cycle. It is a form of fault tree analysis for software that makes many of the errors described earlier easier to find.

Fault tree analysis (FTA) analyzes a system design for failure by looking for program states that indicate an error. It is a common technique in safety or reliability engineering to determine the probability of a safety accident. FTA is not generally applied to software designs. Let me describe a system I built in the early 1970s which used similar techniques in order to achieve high reliability from failure.

PLUM was an error correcting PL/I compiler I developed in the early 1970s [10]. PL/I was a mildly popular programming language initially developed by IBM in the 1960s and 1970s which had a structure that had aspects of both Fortran and Pascal. PLUM's distinguishing feature was that if it found a compilation error, it would generate an error message and "correct" the error. By correction I mean that it would change the input at the point of the error to something that represented a correct program. It might not be (and usually wasn't) the program that the user thought he was writing, but it was semantically and syntactically correct.

This feature was designed during the days when programs were submitted on punched cards and with most compilers, if there was a syntactic error, there would be no useful output and the user would have to wait a day before submitting a corrected program. Using this error correction strategy, even if the correction was not the right one, the process enabled the program to continue to process the program and find other potential problems. And sometimes the compiler even made the correct assumption and did compile the program the user thought he had written.

However, the side benefit of this strategy gave a process for greatly improving the reliability of the resulting program. At the University of Maryland computer center, the staff kept a large file of bad programming cards collected from trash baskets. (This was the early 1970s and keypunches were still being used to type in programs.) This nonsense collection consisting of mistyped Fortran, MAD, COBOL, Assembler and data cards was fed as the input to every commercial compiler on the university system. All crashed! Not only did they fail to find all errors, but they were unable to terminate normally. But not only did PLUM find all the errors in these cards, it even "executed" the program. (We have to take the word "execution" loosely here. For the most part the bad cards were replaced by labels and PLUM simply executed a series of null statements.)

However, what is important here is that PLUM never had a problem scanning for multiple failure modes. Because of PLUM's error correction philosophy, every error found by PLUM was replaced by a sequence of correct PL/1 code. Thus every error was the "first" fault in the program and once it was corrected the program had no remaining faults up to that point in the program. This eliminates the problems described earlier in the SRS and Therac-25 cases. There is never a need for multiple failure modes since every error is the first. And programmers are usually pretty good at fixing initial errors. It is only when keeping track of multiple errors simultaneously that complexity gets out of hand.

Challenges

There shouldn't be a need to justify the importance of computers for the future. Every facet of our society depends upon computer technology. However, I have concerns about how are currently training the next generation of software engineers and how that will evolve in the future. I don't have the answers, so instead I'll give my view of the question. *What do we need to do to get a well-trained workforce in software engineering?* Specifically, how will we address the following concerns in the future, based upon the issues I presented earlier?

1. How to make society realize the importance of software engineering?

As stated earlier, a U.S. government agency does not believe software engineers need to be engineers, yet we have evidence that programs are still hard to write, we continue to make errors, and the increased use of multicore processors will only increase the complexity of future systems. Yet much of society still views programming as just a trade that anyone can do.

A simple anecdote – I have a friend who is an engineer who manages a team that writes systems for astronomers to use. There is also a separate technical support group who manage the computers for the entire institute. Many of the astronomers cannot tell them apart since from the professional astronomer's point of view, both simply make the machines work. However, the technician's role is to monitor the machines, perform backups, and fix system-wide hardware and software problems. They are responsible for the institute wide applications that are obtained from outside vendors, including such products as database systems, word processing and other desktop applications, security products etc. However, the programming staff has the role of designing, building, and testing new application software specifically for the astronomers. Both groups are necessary, but their roles are very different.

A second anecdote – From 1976 through 2002 Victor Basili and I were involved in the NASA Goddard Space Flight Center Software Engineering Laboratory. What I learned over 25 years was that software was never high on a project team's priority list. Spacecraft were designed and then software was considered. Software people were rarely, if ever, part of important decision making panels. The impression I had was that the software people were like plumbing in a building. They were important for the proper functioning of the organization, but were never visible to management.

2. How to create software tools that people really care about?

As a compiler writer, tool builder, and experimentalist over 40 years, I still have my doubts about how successful we have been. The typical conference proceedings today in software engineering contains numerous papers of the form

"How <my acronym>, using <this new theory of mine>,
is useful for the testing of <application domain>
and is able to find <class of errors> better than existing tools."

While this all sounds nice, the tool is never mentioned again in other publications and is quickly forgotten. Few end up as commercial products.

In one study, Dolores Wallace from the National Institute of Standards and Technology in Gaithersburg, Maryland and I looked at various methods used to validate a new technology [12]. We compared the academic researcher and the industrial user. Both groups seemed to have different goals in determining which validation method would be most effective. The researcher wanted scientific validity. That meant controlled replicated studies where most variables could be controlled. That necessarily meant that most such studies were relatively small and that they were done in a laboratory setting and not in the "real world."

On the other hand, the industrial user wants to check out a new technology in a relevant environment. That means large case studies where few of the conflicting variables can be controlled. Replication is rare due to the size and costs of each such development. From the research perspective, these studies are of less value since they are simply anecdotal and do not indicate major trends.

Thus the industrial user and the academic research have differing goals in understanding what is needed when validating a new technology. It is no wonder that many of the academic conference papers do not make it into industrial practices.

3. How do we train future software engineers?

In this paper I gave several examples of how systems have failed. How do we build a relevant curriculum? The long standing dispute of whether software engineering is a separate field or part of computer science has not been settled. While there are some separate software engineering departments, few are part of engineering schools with full engineering credentials for its graduates. Concepts like parallel and multicore programming, fault tree analysis, and project management economics, and measurement and prediction need to be part of the curriculum in addition to the standard concepts of requirements analysis, design, coding, and testing methods.

4. MOOCs?

Finally, we need to address the impact of the massive online open course. Software engineering is basically a laboratory subject, so how to adapt it for an online environment will not be easy. But I believe we must. The economics are forcing the issue and credit courses in an MOOC environment are going to happen. How do we utilize this technology without losing control of the educational process?

All of these questions, except this last one, have been around for years; however, I do not believe they have been solved. We must solve them. Computers control all aspects of our technological world. We must be able to appropriately control the computers.

In conclusion …

Finally, in a paper I recently read from December, 2012: “The developer sets a breakpoint, runs the program, and then sequentially steps through the code from the breakpoint, verifying the state changes as expected.” [13] This is a crude primitive technique which I was able to eliminate in my dissertation in 1971 using a prototype implementation on a 2 megabyte IBM mainframe. Today’s machines can have tens of gigabytes to work with. Haven’t we learned anything in 40 years?

References

[1] Leveson, Nancy G., and Turner, Clark S. (July 1993). "An Investigation of the Therac-25 Accidents," Computer (IEEE), 26(7):18-41.

[2] Neumann P. G. and contributors (March 2011). “Risks to the Public,” ACM Software Engineering Notes 36(2):19-27 (Page 21, Comment by Jeremy Epstein).

[3] Nuseibeh, Bashar (May 1997). "Ariane 5: Who Dunnit?" IEEE Software 14 (3): 15–16.

[3] Grier, David A. (January 2011). “The Migration to the Middle,” IEEE Computer 44(1):1214.

[5] Parnas, David L. (October 2010). “Inside Risks: Risks of Undisciplined Development,” Communications of the ACM 53(10):25-27.

[6] Neville-Neil George V. (December 2010) “Kode Vicious: Literate Coding.” Communications of the ACM 53(12):37-38.

[7] Ortega, Ruben (September 2010). “How Much Software Testing is Enough?” Communications of the ACM 53(9):9.

[8] Sutter, Herb (March 2005). “The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software.” Dr. Dobb's Journal, 30(3).

[9] Ritchie Dennis, Thompson, Ken (July, 1974). “The UNIX Time-sharing System.” Communications of the ACM (17)7, 365-375.

[10] Zelkowitz M. V.(October, 1976). “Automatic program analysis and evaluation, Second International Conf. on Software Engineering.” San Francisco, CA, 158-163.

[11] Basili V., F. McGarry, R. Pajerski, M. Zelkowitz (May, 2002). “Lessons learned from 25 years of process improvement: The rise and fall of the NASA

Software Engineering Laboratory." *IEEE Computer Society and ACM International Conf. on Soft. Eng.*, Orlando FL, 69-79.

[12] Zelkowitz M. V. and D. R. Wallace (1998). "Validating the benefit of new software technology." *Software Quality Practitioner* (1)1.

[13] Spear A., M. Levy, and M. Desnoyers (2012). "Using tracing to solve the multicore system debug problem." IEEE Computer (45)12, 60-64.