Richard Bender ISTQB 2019 Award Acceptance Speech

Richard Bender
Bender RBT Inc.
rbender@BenderRBT.com
www.BenderRBT.com

17 October 2019

I want to thank the ISTQB for this award. The list of prior recipients is very impressive. I have known Capers Jones for nearly 50 years. He was the one who recommended me to be the Technical Lead for the International Y2K Test Certification Standards effort. I knew Boris Beizer well and have known Dorothy Graham for decades. The others, Harry Sneed, Erik van Veenendaal, Robert Binder, and Paul Gerrard, I know of and respect their work and contributions. I am very honored to be in their company.

I want to take this opportunity to make a couple of observations and make a few suggestions about the future of software quality. I began my career in 1965. One of the first computers I programmed was an IBM 1401. It was the size of a very large refrigerator, had 4K of memory, and its speed was about 1,000 instructions per second. It cost $2,500 a month to rent - $21,500 in today's money. Today you can buy a chip with thousands of times more compute power than that for under $1.

In those days there was only one job – programmer. You were the analyst, designer, coder, tester, technical writer, trainer, and user support all rolled into one. Over time, as the technology and the systems we built became more complex, many specialties were created. I moved into full time testing when I joined IBM out of university in 1969.

Actually, the first machine I ever "programmed" was an IBM 407 Tabulating Machine. This was a punched card machine that you programmed by wiring a board. So I started my career with first generation technology. Since then I have had the opportunity to work on systems ranging from those embedded in cars, trains, planes, and medical devices up through applications running on super computers and everything in between. The world of computer hardware and software has evolved faster than any technology in history. I have been fortunate enough to have participated throughout this entire period. It has given me an interesting perspective on the challenges our industry faces. In contrast, it took construction engineering over 5,000 years to go through the equivalent evolution. This rate of change has some significant implications.

The single biggest difference over the last fifty years plus is this: the impact of software defects. When I started, defects were just annoying. Most of the initial systems created were automating tasks that people had been doing manually for decades – like payroll and general ledger. If the system went down, people could backfill in doing the job until the system was fixed. The skills were still available. The impact of defects was, with rare exceptions, contained within the operational walls of the organization.

Today, software is omni-present. Very little happens in the world today without software playing a role. The impact of defects is beyond exponentially greater. Every large organization has multiple multi-million dollars outages per year. In the United States alone, according to a Federal Study done a few years ago, the losses due to software defects were around $300 billion a year. Even worse, there have been thousands of deaths due to defective software. The most common have been plane crashes, with the most recent being the Boeing 737 Max. In that case there was a functional defect related to a single sensor and a usability defect in overloading the pilots with information once the defective sensor went off. Both of these issues could have been found with better testing. In addition, to the degree they were observed by test pilots during development, reporting the issues and/or Management's actions were more than insufficient.

To underscore how severe this problem can be, we just need look at the evening of 26 September 1983. On that night, due to faulty software, the Soviet missile defense systems thought they were detecting incoming ICBM's from the U.S. A full nuclear war was only averted due to the actions of Lieutenant Colonel Stanislav Petrov who overrode the systems. That is just one of a number of such incidents. Software is potentially the most dangerous thing mankind has ever invented.

Most of you will never work on command and control systems so that is someone else's problem. However, many of you work on or will work on business critical, mission critical, and safety critical systems. Wearable devices, including our phones, will continue to integrate into many aspects of our lives. 5G communications will cause tremendous changes in how information is accessed and exchanged. The biggest impact of 5G may be in IoT, increasing the degree of interdependence between systems. In the next decade there will be explosive growth in self driving cars and embedded medical devices. So the stakes have gotten immeasurably higher. Our processes for ensuring quality have to keep up to this challenge.

We do not have the luxury to take 5,000 years to mature like the construction industry. We do not even have the 150 to 200 years that it took electrical engineering to evolve to its current state. We need to do two things if we are going to solve this in a timely manner. We need to narrow the gap between the state of the art and state of the practice. We also need to look at the other engineering professions, especially electrical engineering, to see how they have already solved many of these problems and figure out how to port them over to software.

As an example, I want to focus on just one issue, functional test completion criteria, to show what can and should done.

For white box testing, code coverage monitors are tools that keep track of which statements and branches have been executed during testing. The first such tool was created in 1967 by Herm Schiller of IBM. IBM then established that part of the test completion criteria for all code being shipped from the Labs was to prove you had 100% statement and branch coverage at the Unit Test level and the highest practical level coverage at Integration Test.

This did significantly reduce the residual defect rate. However, after two years of experience, they felt there were still too many code-based defects slipping through to production. In 1969 Earl Pottorff and I were assigned to address this problem. We developed code-based testing algorithms using data flow analysis (latter called set-use pairs). The underlying algorithms were based on compiler optimization theory and the mathematics associated with the "traveling salesman" problem. We found that, even with 100% statement and branch coverage, around 40% of all data flows were not guaranteed to be tested. This is huge gap. Supplementing the test suite to also cover the data flows resulted in finding 25% more code-based defects. It did so, in part, by identifying situations where tests had to be in a specific order, or a test had to be in a specific position in the test stream in order to execute the data flow.

Today there are dozens and dozens of statement and branch coverage tools available for almost every programming language. Yet I rarely see them used. I doubt if even 10% of all the code shipped into production has had its test coverage measured by such tools. When we have gone into organizations and benchmarked their code coverage, it has been rare to see even 50% statement and branch coverage, let alone complete data flow level coverage.

Now let us look at black box testing. High end integrated circuits have a residual defect rate of less than one defect per billions of gates. The residual defect rate at release for software is, at best, one defect per 2,000 lines of executable code – about a million times worse than hardware. The difference is not product, it is process. In creating circuits, they have to write requirements and do design. They actually write code. It compiles into a logic design. Then they test using what is called the path sensitizing algorithms which design tests in such a way that it guarantees that if any logic errors exist, then one or more tests will fail at an observable point. It does this by propagating defects to such a point.

In 1970 William Elmendorf of IBM ported the hardware logic testing algorithms over to testing software from the requirements. He called his approach Cause-Effect Graphing. The algorithms factor in that two or more defects can cancel each out for a given test, resulting in the right answer for the wrong reason. They also factor in that something working correctly on part of the test path can hide a defect in another part of the path. This means that you cannot prove a function is actually working until all of the tightly related tests work in a single run. The tests designed via this approach improve functional test coverage by at least 50% as compared to combinatorics based approaches such as pair-wise testing, all while resulting in smaller test libraries. When we have benchmarked an application's functional coverage, again coverage has generally been less than 50%.

In other words, looking at both the black box and white box coverage, customers and users are doing half of the testing in production, even though solutions have been around for decades.

In many organizations today, even in those producing safety critical software, the actual test completion criteria is that they ran out of time and resources. This leads to what I call "the clean conscious school of software testing" – I did the best I could within the constraints Management gave me, my conscious is clear. Can you imagine a surgeon saying "they only gave me the operating room for two hours so I did the best I could within the constraints the hospital gave me; my conscious is clear"?

I chose these two examples for a couple of reasons. The code coverage monitor has its roots solely in software, demonstrating the need to identify and solve problems unique to our profession. Cause-Effect Graphing, having its roots in electrical engineering, demonstrates a great example of looking to other engineering professions for solutions. Both of these solutions have been around for many years and when applied together do result in systems delivered at or near zero defect. Yet the level of use in the industry is far lower than it should be, underscoring the gap between the state of the art and the state of practice. The fact that so many organizations lack truly rigorous test completion criteria, or do not enforce them when they have them, also underscores why the ISTQB Code of Ethics for Test Professionals is so important. We have a professional and moral responsibility to ensure that the code we deliver is at the appropriate level of quality.

I have spent most of my career working to bring true engineering discipline to the somewhat less disciplined world of software. I have been fortunate to have worked many, many talented people. My most significant mentor was William Elmendorf. I had the pleasure of working with him on Cause-Effect Graphing, on both the automation and the surrounding processes, for many years. I have continued that work with Kevin Ablett. There are many, many others but time does not permit me to mention them all. Though I will contact them and thank them again for their guidance and wisdom.

This brings me back to the ISTQB. Making people aware of best practices and encouraging the industry to develop solutions where gaps exist is a critical part of the ISTQB's goals. The organization and its network of alliance partners have trained and mentored untold hundreds of thousands of test professionals worldwide. I am, again, honored to be recognized by the organization. Thank you.