

August 2007

THE PROFESSION

The Chimera of Software Quality

Les Hatton

Kingston University

Despite years of computing progress, today's systems experience spectacular and all-too-frequent crashes, while many enormously expensive projects fail to produce anything useful. Of equal importance, and potentially more damaging, are the misleading smaller defects we tend to miss.

From time to time, we must remind ourselves that the underlying quality of the software that our results and progress increasingly depend on will likely be flawed and even more dependent on independent corroboration than the science itself. Many scientific results are corrupted, perhaps fatally so, by undiscovered mistakes in the software used to calculate and present those results.

Commercial application areas

I've spent the past 30 years analyzing the quality of software-controlled systems. In every area I've looked at or worked in, often previously undiscovered software defects run rife. In scientific modeling, these defects can lead to highly misleading results. Twelve years ago, with a coauthor, I published the results of a large study of high-quality signal-processing software in the oil industry. Previously undiscovered defects had effectively reduced accuracy in this data from six significant figures to between one and two. However, this data is used to site oil wells and must be of at least three-significant-figure accuracy to perform this task, effectively randomizing the decision-making progress.

We could only discover this because the same software had accidentally evolved nine different times in different companies in commercial competition. Within five years of this, seven of these companies had been bought out or disappeared, so we no longer know the problem's scale, although I hardly think it can have improved simply because we can no longer measure it.

A parallel experiment suggested that similar problems afflict other scientific modeling areas. Sometimes these defects reveal how smoothed our simulations actually are. Thirty years ago, when translating to a sigma coordinate system, I found and corrected an alarming defect in the standard daily forecasting model at the United Kingdom Meteorological Office. The defect zeroed the nonlinear terms in the governing Navier-Stokes equations every other time step, and these terms generate the whole of the weather forecast.

When I reran the corrected model, the differences proved almost impossible to see. Today, we can perform mutation tests to assess this level of sensitivity, but they are rarely used. On too many occasions, an elementary particle physicist here or a specialist in anisotropic wave propagation there has forcefully told me that "our software contains no bugs because we have tested it." This attitude troubles me. I am a computational fluid dynamicist by training, and I know that verifying the science part of any model is relatively easy compared with producing a reliable computer simulation of that science. However, I still can't convince most scientists of this even though I belong to the same club.

Defining a quality scale

Computer science regrettably operates in a largely measurement-free zone. Researchers do few experiments, and even fewer publish their results. Researchers such as Walter Tichy in Karlsruhe have noted this over the years. As a result, software development isn't an engineering industry, but a fashion industry populated by unquantifiable statements and driven by marketing needs. We are exhorted to develop using JavaBeans, OO, or UML because

these technologies will supposedly fulfill our wildest dreams.

This is arrant nonsense. Our experiments to date suggest that by far the biggest quality factor in software remains the developer's ability, and, in most experiments, analysts regularly record variations of a factor of 10 or more in the individuals' performance. This appears to have little to do with any technology or even language they might use. In my experience as an employer, it doesn't even appear to have much to do with their educational background. The best programmer I ever employed started as a 16-year-old with no academic qualifications. Failures in his programs were as rare as hen's teeth. In contrast, one of my worst programmers had a PhD in mathematics. I wish I understood why.

Developers still measure software quality by the number of released faults that have failed per thousand executable lines of code during the software's life cycle. I refer to these as defects. The relationship between this and more conventional engineering measurements, such as mean time between failures (MTBF), remains unknown. However, the best systems appear to be around 0.1 on this scale—exhibiting about one fault for every 10,000 executable lines of code, measured over the software's entire lifetime.

Perfection is not an option, and these faults' effect on the program's output when they fail is unquantifiable. It costs a lot of money to stay this good. My own and other researchers' work suggests that it is at least 10 times worse and possibly as much as 100 times worse in typical computer simulations not subject to the rigorous quality control necessary to stay as low as 0.1.

Nobody knows how to produce a fault-free program. Nobody even knows how to prove it, supposing one were magically provided. I teach my students that in their whole careers, they are unlikely ever to produce a fault-free program and, if they did, they wouldn't know it, could never prove it, and couldn't systematically repeat it. It provides a usefully humble starting point. Some of my colleagues hold out hope for truly verifiable programs, but such methods do not and might never scale to the size of systems we regularly produce.

Much remains to be done, although we have made progress in bounding errors using interval arithmetic. Formalism appears to help in modest ways, as Shari Pfleeger and I reported in a 1997 *Computer* article ("Investigating the Influence of Formal Methods" Feb., pp. 33-43).

Unless we are in complete denial, we know the faults are there but have no methodology to relate the nature of a fault to its ultimate effect on the runtime behavior and results a computer simulation produces.

Computer simulations as proof?

Even in the world of pure mathematics, we are straying toward an era when computer programs become part or indeed all of a proof. The four-color theorem offers an early example of this.

However, computer programs are fundamentally unquantifiable at the present stage of knowledge, and we must consider any proof based on them flawed until we can apply the same level of verification to a program as to a theorem.

Scientific papers are peer reviewed with a long-standing and highly successful system. The computer programs we use today to produce those results generally fly somewhere off the peer-review radar. Even worse, scientists will swap their programs uncritically, passing on the virus of undiscovered software faults.

A widespread problem

In my experience, industry probably fares better because it uses successful test procedures more widely than does academia, which normally cannot afford the degree of verification necessary to reduce defects to an acceptable level. Even so, the world is rife with software failure. My TV set-top box crashes about every seven hours, according to my records, shutting itself off in about one in three cases. It's a piece of junk.

My PC packages crash frequently as well. Updating my gas meter reading on the brand new British Gas telephone entry system failed the first time and accepted the same reading the second time, having successfully repeated it back to me both times. Even worse, this system appeared to leave open the possibility of changing somebody else's account details. When trying to register for the annual British Computer Society Lovelace award ceremony, the payment site had accidentally been deployed in test mode, meaning no money would change hands.

Looking farther afield, the automobile industry has begun to suffer extensive recalls based on software failures affecting all electronically controlled parts of the vehicle, including but not limited to the brakes, engine management system, and airbags that made the *New York Times* in 2005 (www.nytimes.com/2005/02/06/automobiles/06AUTO.html).

The cost of poor quality

Poor software quality affects us in other ways. If the technological nations really understood how much money developers throw at failed software projects, they would join in an international outcry. In 2004, the UK Royal Academy of Engineering made an authoritative case in a comprehensive report after interviewing many experts. Despite this, the initiative—as far as I can see—stalled, afflicted by the peculiarly widespread laissez-faire attitude that attends anything to do with computers. People simply do not appear to care enough.

Yet the amount of money wasted likely falls between £10 to £20 billion per year in the UK alone. In 2002, the National Institute for Standards and Technology produced a hauntingly similar conclusion in the US. Quite recently, several of my distinguished colleagues wrote to the *London Times*, stating the case for an independent audit of the deeply troubled \$25 billion UK National Health Service's Connecting for Health project. It was rejected. At the same time, I interviewed 10 disparate members of the NHS at random and received unanimous and deep concerns about this system's quality and relevance.

I've analyzed enough failed systems in my time to know the two classic symptoms of a system on its way to the fairies. First, no independent audit is allowed, and, second, talking heads tell you everything is fine when the ultimate users tell you the opposite. Ironically, as I wrote this line, my word processor crashed, probably in sympathy. Everybody should have a law, so here's mine

The technological societies will collectively trash around \$250 per person per year on systems which will never see the light of day or, if they do, do not come close to what their users wanted, assuming they were asked in the first place. This they will ignore.

Room for optimism

Not all is bleak. Personally, I feel optimistic. The idea of fully reproducible research, originally proposed by Jon Claerbout at Stanford, is an important step in the right direction: The science and the complete means to reproduce the computational results are packaged together to extend the highly successful peer-review system to the software as well as the science.

The Linux kernel is now arguably the most reliable complex software application humanity has yet produced, with an MTBF reported in the tens and, in some cases, hundreds of years. Poetically, the Linux development environment, which leverages the contributions of thousands of Web volunteers who give their spare time for the public good, breaks just about every rule that software process experts hold dear.

Furthermore, Linux is written largely in programming languages that cause palpitations in many language experts. Despite this—or, who knows, even because of it—the open source community has demonstrated that it is perfectly possible to produce extraordinarily reliable software. This same community has created many other examples of highly reliable applications. We really ought to understand this better than we do if we are to be worthy heirs to an engineering legacy.

The accumulating evidence shows that most software failures and disasters afflicting us today could have been

avoided using techniques we already know. They affect everybody and should not be ignored. In a scientific context, they undermine the very fabric of our work, so must we really continue building scientific castles on software sands when we could do so much better? I hope not.

Les Hatton is professor of Forensic Software Engineering at Kingston University, London. Contact him at L.Hatton@kingston.ac.uk.

*Editor: Ne4ville Holmes, School of Computing, University of Tasmania; neville.holmes@utas.edu.au.
Links to further material are at www.comp.utas.edu.au/users/nholmes/prfsn.*