21st century Software Engineering: Largely cloudy with occasional bright periods. Rain expected.

Les Hatton

First of all, I would like to thank Felix for inviting me to write an article but most of all for his terrific efforts of the last couple of decades in editing this magazine and generally trying to keep the momentum going in safety-critical systems.

I paused my own scientific career around 25 years ago to study software defect. Given its impact, I might as well have studied rheumatism in beetles but I'm not sorry as I have observed or discovered lots of interesting things about systems generally. To my sorrow however, the enduring lesson has been that very few people actually care about software failure until its too late. Even with a good system, which engineers would naturally try to make better for the same price, their management want to make it 'as good' but cheaper. This is management speak for 'cheaper' – they have no idea what 'as good' even means, and I'm not entirely sure we as software engineers do either. We don't do prevention, so I've given up and gone back to science on the grounds that it is slightly less masochistic. I don't think I can stand the Groundhog Day experience of rummaging round in the bowels of yet another failed system.

You might dismiss my assertion above as the febrile wanderings of an old curmudgeon so let me explain why I don't think we care enough and why, as a result, engineering in software is often dismal and frequently non-existent. The best engineering has always evolved by understanding its source of defect through careful measurement and analysis and systematically eliminating them by novel designs and/or implementations. It takes time and patience but its not rocket science. Unfortunately, software engineering is in far too much of a rush to bother with this simple but reliable process and we are simply drowning in code, paradigms and processes as a result. It won't make me any more popular but it seems to me that academic Software Engineering has had very little beneficial effect on the systems we actually have to use. Let me pick a few pressure points just as a backdrop.

The automotive industry seems hell-bent on stuffing as much software as possible into the humble car, whatever its perceived value, and there now appears to be tens of millions of lines of code squirrelled away in numerous systems, (Mossinger (2010) – and one of a number of articles in the Software Impact section of IEEE Software reporting gigantic systems). Not surprisingly, there are now frequent recalls (just google them) and a glance through the supporting documents of some of these, particularly the ones that reach the courts, reveals a surprising lack of awareness of important principles of software engineering, and reliance on dubious and ancient metrics such as cyclomatic complexity. This great outpouring of creativity has now sprouted new heads like a latter-day Hydra. A few months back, a car was hacked by professional hackers through its entertainment system and reversed into a ditch, (they told the driver first so he could get out). At the same time, VW added the software cheat to the software engineering lexicon, but don't seem to know where it is or who put it there. The last time I looked, there seemed quite a lot of software in a car which could be called safety-related but I must have been mistaken. Silly me.

Speaking of security, at the time of writing, Talk Talk appear to have lost some 1.3 million customer records to hackers (at the latest count) through a SQL-injection attack. One of the first things we used to teach students in university when soliciting data in web-sites was to protect against these. It's not as if we haven't seen them before and I am frankly amazed that we have this level of incompetence in the 21st century. They are of course not alone. Loss of personal data is essentially

an epidemic. We either give it away, lose it or have it taken away. Of course you don't need software to lose personal data but it certainly helps when you want to lose lots of it and most personal data loss seems to involve sloppy software engineering in one way or another.

Requirements seem still to be very unevenly applied. As a result of the ubiquitous sensor-riddled smartphone, anybody can be found anywhere on the planet to be sold useless junk, but with bitter irony, we can still lose a 350 ton aeroplane somewhere in the Indian Ocean. Perhaps all systems are doomed to be evolutionary in some sense as we never seem to know what we want at the start, and subsequent fiddling with a few million lines of code which has incrementally sprouted in order to add a few more new features or even fix what we thought we were trying to do in the first place, remains a profoundly error-prone process. I sometimes hear talk of 'self-healing' systems and other wonders of the abstract, but I shouldn't have to remind readers that a swamp too, is self-healing.

Software has been of great benefit in the latter part of the 20th century but we are in danger of losing really important concepts as it continues to permeate everywhere like bindweed. Perhaps the most serious loss in the long run is that of Popperian reproducibility in science. This has been a personal hobby horse of mine for perhaps 25 years and is why I started studying software defect in the first place. The very essence of the scientific method is independent reproducibility. If a result could not be reproduced independently, there was a time when it was ruthlessly discarded as it must, to make sure progress is based on robust results, but not any more. The influx of massive amounts of software and computation into most sciences has added a new and unquantifiable layer of opacity. As a result, very, very few scientific results with significant amounts of computation are reproducible. Even the most famous scientific journals such as Nature and Science struggle with this and still fall far short of enforcing it, relying instead on rather vague and ineffective concepts such as 'code-sharing'.

I really don't believe this can be over-stated so I give it a sentence to itself. *If a scientific experiment involving significant computation is not accompanied by the complete means to reproduce those results, then it is NOT science. Anything short of this is unacceptable.* Of course the problem here is that even the most basic of software engineering concepts, such as careful revision control and designing software to be testable, are generally not taught to scientists. Putting together a complete reproducibility package is not difficult but it is a little time-consuming to the scientist in a hurry, (Hatton 2015). This example allows each diagram, table and statistical result to be reproduced from an information theoretic study in post-translational modification of amino acids in proteins including regression tests, all source code and the means to run it, starting with a download of a 2.7Gb open access protein database; open data => open source => open reproduction.

One other area where I see problems developing rapidly is the alienation of many users from the systems they have to use. As management grabs any opportunity presented by software to reduce the number of people they have to employ, those people are replaced by some simply awful systems designed by programmers who do not appear to know what a human being looks like, let alone behaves like. Today I walked into my local Barclays to be greeted by 2 new machines which have recently replaced counter staff, but necessarily attended by one dramatically over-worked staff member whose job was to explain to the customers how to work them. One of their engaging features was that the screen buttons were slightly offset from where you pressed if, as I am, you are tall. The queue was gigantic for the other staff member allocated to handle the numerous things the machines couldn't, but no less so than that attempting to check in to Easyjet at Gatwick on the way to Lisbon on holiday last week.

There, every automated check-in machine suddenly decided to refuse luggage without any explanation whatsoever, again with frantic staff running up and down trying to get the machines to

function by over-riding them with swipe cards. The week before it was the turn of BA at Heathrow on my way to Germany where the first two online check-in machines refused to check me in, issuing the now traditional incomprehensible messages, and the third said I had already checked in and refused to issue a boarding pass. I am long past the days where I used to march up to the check-in desk and report in vivid detail exactly what went wrong and where I thought it was. Now I just slink up, say I'm jolly old and grin apologetically. Many years ago, the doyen of software testing, Glenford Myers, told me that it's not going to get any better unless we tell the supplier how awful it is. Sorry, Glen, I tried, I did and it didn't. The suppliers have neatly side-stepped this anyway by removing anybody human you can talk to and making you complain to another awful bit of software that their first bit of awful software is well, awful. Sigh.

There are occasional bright periods and I still get a buzz from using a really good piece of software. I am terrifically impressed by the exemplary reliability of important communication components in the web, mail handling agents like Postfix which I use a lot, transport layer security components and many others too numerous to mention, and I have had nearly 20 years of what lawyers euphemistically term 'quiet ownership' in using the Linux kernel. These are all open source, which as the flag-bearer for computational reproducibility are happily the potential saviour of the scientific method.

We and generations to come owe a debt of thanks bigger than we can possibly imagine to open source software but how does this fit in with software engineering ? I really have no idea. Do we still actually do software engineering ? If so, why do we produce so many dreadful systems or is time for my medicine again ?

References

Hatton, L. (2015) “Reproducibility package for pone_0125663.html”, http://www.leshatton.org/pone_0125663_reproducibility.html

Mossinger, J.M. (2010) “Software in Automotive Systems”, IEEE Software, 27(2), p.2-4

Les Hatton is Emeritus Professor of Forensic Software Engineering at Kingston University. He also has a shed in which he applies Information Theory to try to make sense of enormous protein databases. Yes, really. lesh@oakcomp.co.uk