Balancing safety with rampant software feature-itis

Les Hatton

SEC, Kingston University and Oakwood Computing Associates Ltd, UK

Abstract *In the 30 years or so that we have been developing a safety methodology to accompany the growing presence of software in safety-related systems, understanding of the software development process itself hardly seems to have advanced. We still use in most part the same languages we did then but duly bloated to match the uncontrollable growth of software itself. We still teach entire paradigms without any basis in the scientific method whatsoever whilst the amount of software continues to grow alarmingly, particularly in the automotive industry. The result is an absence of any real forensic basis for understanding failure and subsequently avoiding it. This short essay looks at some of the reasons why, and demonstrates from recent results in information theory, that this stems from the fact that we are probably barking up the wrong tree.*

1. Introduction

30 years ago, 50,000 lines of code was considered pretty big. Ian Sommerville's magnum opus on Software Engineering had just appeared (Sommerville 1982, 2015); university computing departments formerly known as maths, electrical engineering, chemistry or indeed anything vaguely scientific were rushing to teach students the new ideas as computer science; the early versions of Unix had appeared embodying a new all-embracing language C; we taught compiler design, discrete mathematics and other hard subjects; the first twinklings of object-oriented design were stirring in the darkness beyond the Styx; we thought we might be able to prove programs were correct, perhaps even routinely; we were beginning to realise that developing software for reliable systems was actually extraordinarily difficult; AND we also started measuring properties of software such as how many branches or how many *goto* statements it had, in order to make ambitious and entirely unsupported statements about the relationship of defects with these so-called metrics, (for which I must bear my own share of guilt).

Today, we encounter systems with 50 million lines of code, (Software Impact series, starting IEEE Software 2010). Ian Sommerville's 10th edition of Software Engineering has just appeared (Sommerville 1982, 2015); university computing departments are reconsidering returning to maths, electrical engineering, chemistry or indeed anything vaguely scientific to continue to pull in the punters[1]; Linux written in C along with countless open source packages runs the internet more or less (and very well too thank goodness); we mostly teach people how to do websites and play with SQL databases; we have given up with object-orientation and have renamed everything in terms of sporting analogies, so we have scrums and rushes and we have formed agile alliances and movements, and soon we may have television shows called "Programmer's partners" (I made that bit up); we haven't the faintest idea how to prove 50 million lines of code correct, (whatever that means when the requirements change more quickly than the verification time); we have realised that because nobody wants to write programs any more we are desperately short of programmers and now advertise via the BBC and other such paragons of science, "learn to code in a weekend"; AND we still extract endless largely meaningless metrics in the thin hope that they will correlate with something useful other than inside leg measurement.

What we still don't have is any systematic technology to understand why and how software based systems fail, so where did the wheels come off the great Software Engineering Juggernaut, and why are we now wallowing in millions of lines of the stuff which needs updating (i.e. fixing) on an often daily basis, and most of which affects our lives?

2. The rise and fall of Software Engineering

The whole problem probably started, certainly in the UK, around the time we began to conflate Computer Science in universities with ICT in schools, (Information and Communications Technology also known as "Computer Skills") such as getting a spreadsheet to add two numbers; writing a sentence or two in a word-processor or designing a web page). ICT was intended to titillate the palates of our young, preparing them for careers in Computer Science. In truth, it probably bored them senseless. Perhaps worse, ICT spread into management offices. Even the CEO had a computer on his or her desk and could fiddle about with a spreadsheet, no doubt thinking how jolly easy it is to change things. In the army, they say there is nothing so dangerous as an officer with a map. In business, it is a CEO in charge of a spreadsheet.

Meanwhile Computer Scientists were largely having far too good a time inventing new stuff to lay down the basis of a scientific and engineering methodology based on measurement-based systematic improvement for software development.

[1] At GBP 9,000 – 12,000 per year in England and Wales for example.

If a structural engineer says “if you build a horizontal beam that long, it will break”, they have decades of measurement-based material science to back it up. If a software developer says “If I add that feature, it will probably cause serious unintended consequences”, they have nothing to back it up other than their experience, so it's very easy for a CEO aka spreadsheet jockey, to override them.

We therefore seem to have come full circle and arrived at a paradox. We are rushing to produce a new generation of “coders” as a proxy for proper engineering to crank the stuff out, whilst we are simultaneously overwhelmed by millions of lines of code in systems such as the formerly humble automobile, much of which in my opinion we do not fully understand, given the levels of reported failures. We have rampant feature-itis – not everywhere it's true, but in too many uncomfortable places and it seems to me that the distinction between what we used to know as safety-related software development, and the systematic pumping out of millions of lines of code for the latest in-car gimmick or whatever, has blurred to the point where there seems almost no distinction.

3. The growth of feature-itis

Software is mostly sold like soap powder as part of a fashion-based industry. As I have many times discovered to my cost in commercial software development, the moment you release a brand new version of your software with lots of ravishing bells and whistles, the first thing the users tend to say is, “Why can't it do this …?”. I have always felt that this is tantamount to buying a television and then wondering why it can't tumble-dry. It is symptomatic of the widely held delusion that software is infinitely malleable, whilst maintaining its integrity. One out of two isn't bad I suppose ...

This has led to rapid growth in many software-producing industries as suppliers compete to please end-users with new features. The bottom line is that most systems are now growing in source code terms at around 20% per year with a surprisingly small variance, (van Genuchten and Hatton (2012)). This is roughly equivalent to a doubling in size every 42 months. No wonder we are wallowing in it and Fred Brooks' prescient quotation from Ovid made 40 years ago, “add little to little and you finish up with a big pile” (Brooks 1975) is perennially true.

I didn't expect to see this in safety-related systems however. I thought we would have a little more sense and that our burgeoning safety bureaucracy with its safety cases, safety management, too-often impenetrable safety vocabulary and plethora of standards such as IEC 61508[2] would have acted as a brake on vaulting over-ambition. It hasn't. Take the automobile for example. When I first started working in and around this industry in the early 1990s, there were around 100,000 lines of assembler in a car. 20% a year for say 23 years gives 12.8 million lines

[2] http://www.iec.ch/functionalsafety/

give or take a useless feature or three, and that's about where we are today, (Mossinger, 2010), except that it's mostly C now. It is true of course that it is not all safety-related but a significant amount is and the demarcation line between the bits which are safety-related and the bits which are not remains fuzzy to say the least, particularly as they tend to be distributed throughout multiple shared processors.

As a result of this unbridled creativity, we now have in 2016, the first reported death due to automatic driving when a Tesla system could not distinguish a large truck from a light sky, (Tesla (2016)); there have been 149 recalls involving tens of millions of cars (PopSci, 2016) and including unintended acceleration incidents, braking failures, engine cut-outs and numerous other features specifically related to software; we have drive-by hacking of cars via the stack of internet-related software we are now serving up to drivers for their comfort and "safety" (BBC 2013); and we have the software cheat, software designed solely to mislead, courtesy of Volkswagen and probably numerous others. In fact as I write this part on 9th September 2016, the BBC news had two items: 1) GM were recalling 4.3 million vehicles because of a software defect which might cause airbags not to deploy, a situation that may already have led to one death. It is expected to cost about half a billion dollars, (GM 2016). 2) In contrast, the success of a robot arm in improving surgical precision was reported, so clearly some systems not so prone to feature-itis are benefiting, (BBC 2016). I sincerely hope that the designers of the robot arm are not considering internet-stack upgrades so the surgeon can play the latest hot-off-the-shelf exercise in mindless violence in between patients, to keep their reactions up to speed.

Apart from the odd billion dollar pay-off to avoid further litigation, (Toyota (2012)), the legal profession has remained relatively quiet about software, largely because they are in the dark about it, just like the rest of us. The probable reason for this is that the almost complete lack of any empirical basis to software engineering means that expert witnesses can violently disagree with each other without perjuring themselves, (30 years ago or so, the courts even went so far as to comment on this in *Saphena Computing v. Allied Collection Agencies* (1985) – plus ca change ...).

4. Software development in court

A particularly high profile example of software problems in automobile systems is Toyota, (NASA (2011), Barr (2013)). These reports are very detailed and thorough. However, it becomes apparent that there are areas of coding and coding practice which are crying out for more substantial measurement support. For example, the use of global variables is an important issue in these reports as a dangerous practice. It is likely that many embedded system engineers would agree, but how dangerous are they and in what context are they dangerous? Everybody

will have an example which may have affected them but this does not constitute a quantitative predictive system. Tim Hopkins and I checked this in the NAG library (Hopkins and Hatton (2008)), and there was no statistically significant relationship between the presence of global variables and the presence of defect after 25 years of use. Although hard real-time systems and scientific subroutine libraries are not the same beast, there is precious little sharing of good experimental data and analysis in any computationally reproducible form on which we could build. It could be argued that there are better, more modern alternatives, in that they make more sense to an experienced engineer, but where is the quantitative evidence that they are "better"?

There are other examples quoted in the reports associated with coding rules but again, even though we can compare their transgression rates with other systems, it tells us very little about any reliability implications. In fact, predicting reliability generally is pretty difficult, (Littlewood and Strigini (1993)). Reading the above reports as an engineer, I agreed entirely with the verdict but not with some of the arguments presented to support it.

(Note that Toyota have since settled with the US Justice Department for a reputed 1.2 billion dollars (Toyota (2012)). That would have paid for a lot of testing.)

5. Why is measurement-based improvement so difficult?

A good question. If we consider this in terms of control-process feedback, one important component of an answer seems to be that of disjoint timescales. In the time it takes to understand the failure modes of a process so that it can be changed beneficially, it is very important that the process has not changed significantly whilst analysis is in progress. This is rarely the case in software development, where there is such a great creative outpouring of paradigms, methodologies and other distractions that empirical support for these ideas simply never happens.

There is another and much more fundamental problem. What is it that we have to measure to bring about beneficial feedback? In control process feedback, the general idea is that we identify systematic modes of failure of the resulting product and then we change the process which produced it, either to mitigate or to remove such failure modes. This might be through either an improvement in the product itself and/or by use of some design methodology such as redundancy. Unfortunately, this relies crucially on successful diagnosis of failure, notably the ability to relate cause to effect in numerical terms so that we can quantify the risk of failure of certain practices.

In this we seem to have failed abysmally. Raking through a few hundred thousand lines of code and perhaps a lot more, trying to pin down certain kinds of failure turns out to be extremely difficult, as demonstrated by NASA (NASA (2011)), who tried but failed to find a specific software defect which led to the Toyota un-

intended acceleration failure, whilst later, Barr (Barr (2013)) did. Since it is much easier to see source code than systems behaviour, (which we view through the very dark glass of testing and the even darker glass of user experience), we have become fixated with source code metrics – how many lines, how many decisions, how many loops, how many variables (global or otherwise) – in pursuit of the dream that we might be able to predict defects from some combination of these measurements. In the process, we have spawned derivative candidate measures such as function-points. Indeed most of the literature on software defects seems to be attempting to fit some kind of statistical model based on lines of code, number of decisions, fan-in fan-out or whatever. It was hoped that this might allow us to define what we mean by software quality and be able for example to distinguish good code from bad code, which would at least be a start.

Not only have we failed, but it seems from recent theoretical studies that we cannot succeed in this endeavour. Indeed, software systems are one of a very wide class of discrete systems, (systems built from components, which are themselves build from indivisible discrete pieces), which conserve Hartley-Shannon information (Hatton (2014), Hatton and Warr (2015)). This conservation principle gently acts underneath the frenzied creativity of programmers, whatever language they are using and whatever application area they are working within, to force the length distribution of components – functions, subroutines, packages and the like – into the same characteristic canonical length distribution, (shown as Figure 1 for 80 million lines of C). Its shape is very interesting and comprises of a rather complex sharply unimodal distribution of lengths with a characteristic and extraordinarily accurate power-law for longer components and is the implicit solution of:

$$t_i = A.exp(-\frac{1}{2t_i}).(1 - a_i/t_i)^{\beta}.exp(\beta\frac{a_i(1 - a_i/t_i + 1/2t_i)}{t_i(1 - a_i/t_i)}).a_i^{-\beta}$$

where A, β are undetermined Lagrangian constants, t_i is the length of the i^{th} component and a_i is its frequency, (which turns out to be proportional to its unique alphabet).

In other words, our fixation with the size of components and their possible relationship with defects is almost certainly misplaced. Indeed, we do not seem to have any real control over the length distribution, instead the conservation principle takes over, as it operates at all scales.

This has a number of important implications. First of all, the average size of components is conserved around the mode of this distribution. Second, it would appear that the distribution of defects across software systems is purely statistical. When defects are randomly distributed amongst component sizes with this distribution, this has the non-intuitive property that a large percentage of software components in any system, something like 75% will never exhibit any defects, and leads to the frequently-observed phenomenon we know as *defect clustering*.

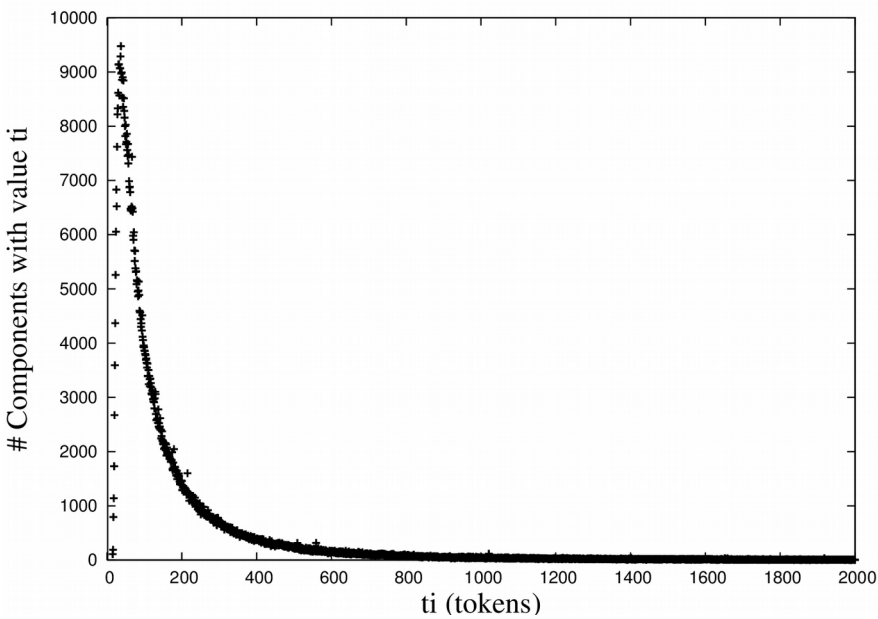

Fig. 1: The canonical length distribution of all software demonstrated in a large open source population of C.

On the face of it, this sounds like a vision of the Holy Grail and if only we could understand how these apparently defect-free majority were built, we might conceive of building a (near) perfect system. Unfortunately, they look just like the rest and it appears to be due simply to the random spreading of defects amongst components which have the canonical length distribution mentioned above. Whilst there is some evidence that we can exploit the resultant defect clustering by identifying a defect and then utilising the increased probability of finding another one, understanding exactly why a component has been defect free for a long time appears to be as futile as understanding why some people never win the lottery by studying people who previously have. It is a purely statistical effect caused by the underlying conservation principle - the reason that the majority of components never show any defect appears to be simply because the rest do have defects and they have to be somewhere.

6. Where now?

It is easy to criticize but we must be constructive. Society depends so much on software now that it would be foolhardy to blunder on regardless, although I see

no immediate end to the feature explosion in sectors like the automobile industry unless there is a corresponding backlash from the end user. It costs a billion dollars or so on an increasingly regular basis, but the car-industry seems to be profitable enough to take the hit[3], and as even embedded software becomes more and more easy to change over the internet, updates come increasingly often, as there is little or no logistical barrier. This may be very well for the almost daily Android software updates, but the relationship between "little and often" and safety-relationship has yet to be explored.

The emerging results from information theory are a little more worrying, although they do explain why our efforts at producing predictive metrics of defect from properties of source code and the processes that produced it, have not yielded the reliability gains we might have hoped for. In summary, it seems to me therefore that *if we are to make progress in improving the safety and reliability of systems containing software, we are rather more likely to make it by pursuing the traditional engineering virtues of redundancy and designing for failure than we are by pursuing studies of the fabric of software, which, as far as information theory is concerned, is just a bunch of symbols*.

References

BBC (2013) http://www.bbc.co.uk/news/technology-23443215, accessed 11-Oct-2016.

BBC (2016) http://www.bbc.co.uk/news/health-37246995 accessed 09-Sep-2016

Barr M, (2013) http://embeddedgurus.com/barr-code/2013/10/an-update-on-toyota-and-unintended-acceleration/ accessed 10-Sep-2016.

Brooks F.P. (1975) "The mythical man month", Addison-Wesley, ISBN 0-201-00650-2

GM (2016) (http://www.reuters.com/article/us-gm-recall-idUSKCN11F2AH, accessed 09-Sep-2016)

Hatton (2014) "Conservation of Information: Software's hidden clockwork?", IEEE Transactions on Software Engineering, 40 (5), p. 450-460.

Hatton L. and Hopkins T. (2008) "Exploring defect correlations in a major Fortran numerical library", http://www.leshatton.org/NAG01_01-08.html

Hatton L. and Warr G. (2015) "Protein Structure and Evolution: Are they Constrained Globally by a Principle Derived from Information Theory?", PloS ONE, doi:10.1371/journal.pone.0125663

IEEE Software (2010) "The Software Impact columns", ed Michiel van Genuchten and Les Hatton, 2010-

Littlewood B. and Strigini L (1993) "Validation of ultrahigh dependability for software-based systems", CACM, 36 (11), p. 69-80.

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

NASA (2011) http://www.nhtsa.gov/staticfiles/nvs/pdf/NASA_FR_Appendix_A_Software.pdf, accessed 10-Sep-2016

PopSci (2016) http://popsci.com/software-rising-cause-car-recalls, accessed 29-Sep-2016

Sommerville I. (1982,2015) "Software Engineering", Pearson, ISBN 1292096131 (1st edition 1982, 10th edition 2015)

Tesla (2016) https://www.theguardian.com/technology/2016/jun/30/tesla-autopilot-death-self-driving-car-elon-musk, accessed 29-Sep-2016.

[3] "A **billion here, a billion** there, pretty soon, you're talking real money", as the celebrated Republican Senator Everett Dirksen was alleged to have said. Never a truer word ...

Toyota (2011), (http://www.wsj.com/articles/SB10001424052702304256404579449070848399280, accessed 10-Sep-2016)
Van Genuchten M. and Hatton L. (2012) “Compound Annual Growth Rate for Software”, IEEE Software 29 (4), p. 19-21.