"Repetitive failure, feedback and the lost art of diagnosis"

by

Les Hatton[1]

Version #1.4, 22nd. November, 1998

Abstract

This paper highlights a growing problem as software systems become more and more tightly coupled and complex, that of poor diagnostic capability. Diagnosis has never been particularly sophisticated in software systems, often being an ad hoc process in which programmers receive no training. As a result, significant numbers of failures in modern systems cannot be diagnosed in the sense of being uniquely related to one or more faults and as such they continue to fail. As a result, modern software systems are unique amongst modern engineering systems in being characterised by repetitive and frequently avoidable failure.

This paper discusses the strongly related issues of repetitive failure, feedback and diagnosis in a generally light-hearted way but makes a plea for diagnosis to be an essential feature of systems design. All engineering systems fail and this knowledge should be part of the software design process such that inevitable failures can be quickly related to the contributing fault or faults and the system corrected to avoid future re-occurrence.

---

[1] Oakwood Computing, Oakwood, 11 Carlton Road, New Malden, Surrey KT3 3AJ, U.K.
lesh@oakcomp.co.uk and Computing Laboratory, University of Kent, UK

The nature of repetitive failure

Its a great privilege to be asked to write this essay for JSS and I thought that I would take the opportunity to stand back a little from the fine details of modern software engineering, (as if I understood them …), and talk about engineering in general and how little its lessons seem to have been absorbed in mainstream software engineering.

We stand at a cross-roads today. Software engineering produces very unreliable products compared with hardware and this is amply evidenced by the behaviour of the average PC. Hardware failure of such devices is very rare indeed with hard disk-drives enjoying MTBFs of up to 1,000,000 hours and other parts of the system sharing similar reliability. In comparison, the software is appalling with most packages exhibiting failures of one kind or another every hour or so, (Hatton 1997a). In other words, there is something like five orders of magnitude difference between the hardware reliability and the software reliability. For example, just two weeks ago, in porting a simple and very reliable C program from the devastatingly reliable Linux environment which I have *never* had to reboot, to the latest Borland C++ Builder environment running on a modern PC hosted by the latest "service pack" of Windows'95, I had to reboot the machine 16 times during the day, none of which was due to my program - I didn't even get the chance to run mine until the day after as I spent much of the day watching the Windows '95 start-up screen. This isn't unusual, there just isn't that much difference between the competing products on this platform. We probably should expect this as PC products are not sold for reliability, they are sold and bought on fashion grounds for feature sets only, an issue I will develop shortly.

This would generally not be a problem but their ubiquity now encourages people to use them where reliability is crucial. Furthermore the rest of our software efforts do not on average produce very much better systems. We seem to have learned nothing in the last twenty years and in some ways have gone backward. I would like to quote the following words:-

"Our students graduate and move into industry without any substantial knowledge of how to go about testing a program. Moreover, we rarely have any advice to provide in our introductory courses on how a student should go about testing and debugging his or her exercises."

“Every programmer and programming organisation could improve immensely by performing a detailed analysis of the detected errors, or at least a subset of them”.

“An efficient program debugger should be able to pinpoint most errors without going near a computer”.

Most software engineers and engineering managers would agree with these fine words, but it comes as something of a shock to realise that they were written in 1979 in “The Art of Software Testing” by the doyen of testing, Glenford Myers, long before we invented ISO 9001 and the CMM. Nothing has changed as far as I can see, except the size of the systems we produce and the sensitivity of the environments in which they run, both of which have increased dramatically. If anything, there is *less* focus on testing now than there was then. Its as if we’ve given up. Instead the magazines, journals, conferences and so on exhort us to try a bewildering and ever-increasing array of new technologies with no measurement support whatsoever in pursuit of the modern goal of ‘reduced time to market’. For example, in spite of the huge investment in OO techniques, such data is there is, (Hatton 1998), is very ambivalent over the claims. The problem of course is that we cannot distinguish the fashion element of computing science from its engineering element and in general when an article says something like “buy this, its good for you”, what it really means is “buy this, its good for me”.

The notion of ‘reduced time to market’ is worthy of expansion. What we are trying to say is how much failure can we get away with by releasing the product before it is really ready in order to garner more market share. This embodies the classic risk-benefit trade-off as depicted in Figure 1. (You can add the fashion notion of features into this argument also but to keep it simple I won’t). This trade-off simply states that the more testing you do, the better the product but the later the time to market. Unreliability is a commercial risk which can damage the developer and early time to market is a benefit leading to more market share. The most important factor about this diagram is that it illustrates that *the amount of testing to be done is an economic decision made by the developing company’s management*. The decision when to stop testing is palpably *not* an engineering decision although in those companies without such measurement control of testing, it all too often is. (Note also that many products fail to achieve a nice clear-cut curve like this and its shape can be an excellent process diagnostic !). For example, it is common to release PC products early on this curve because reliability is not currently a big development risk because the fashion basis of the PC world dictates that only time

to market really matters a lot. On the other hand, a nuclear reactor control system would hopefully be rather more to the right, although it must be remembered that even those systems have a market opportunity so will be released before they achieve perfection. *Everything* is released before perfection because we can't afford to wait long enough. This is a general truth about engineering.

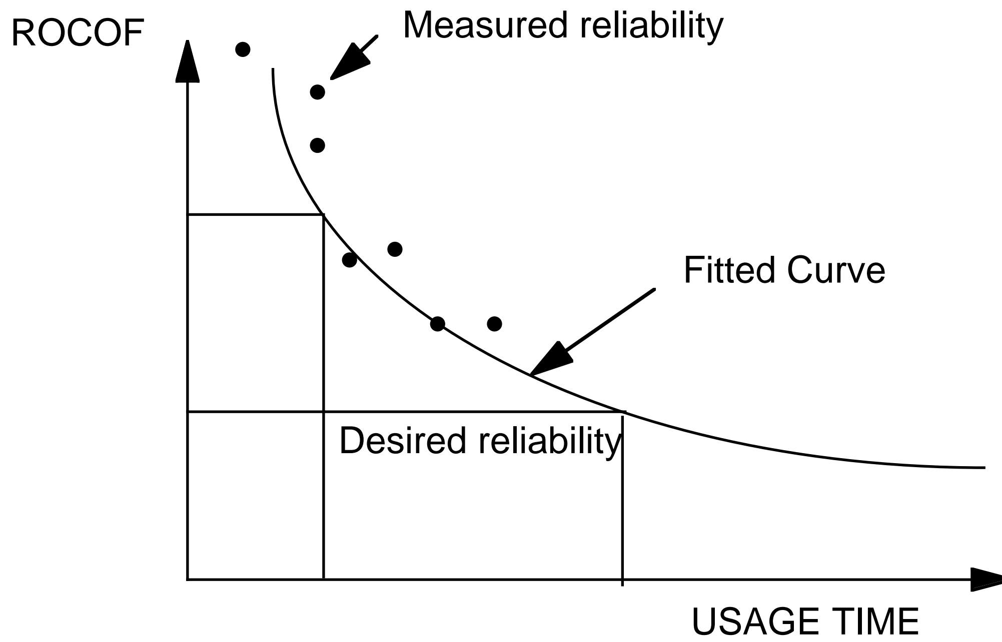

Figure 1: A classic reliability growth model exemplifying the trade-off between risk, (the y-axis), and benefit (the x-axis). We are measuring here the Rate of Occurrence of Failure (ROCOF) against the usage time of the system. (Hopefully), as the system is used, its reliability increases as its failure rate decreases with defect correction until it reaches some acceptable level of reliability, at which point it can be released.

We have now seen the fashion element at work, but what should the engineering element be based upon ? To put it in its simplest form, *the essence of engineering improvement is based around the elimination of repetitive failure*. Glen Myer's middle quotation above embodies the spirit of this. Since the dawn of engineering, centuries ago, the standard method of engineering improvement has been epitomised by Figure 2:

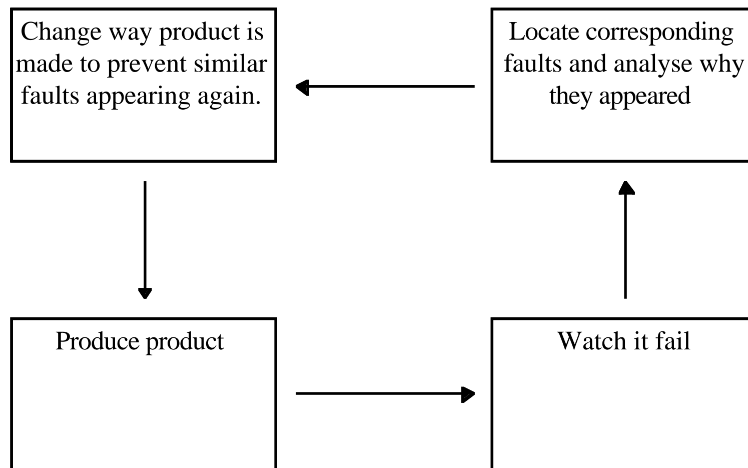

Figure 2: The standard method of engineering improvement over the centuries

This should evince some excitement. It is the silver bullet we have been seeking all along. Rather inconveniently, it is based on sound engineering practice and hard work, rather than some mystical process which can be done by anybody with a mouse. It is guaranteed to drive a product in an improvement direction more or less independently of intuition, which on its own is generally a very poor guide. It is not guaranteed to find the best method however, although it does provide a measurement and analysis framework to recognise a "tired" technology. This of course is analogous to the difference between local and global optimisation, ((Fitzpatrick, Gelatt Jr et al. 1983), but also have a look at the wonderful book of (Dawkins 1996) for a biological analogue to this). In local optimisation, the metric for improvement, in this case faults injected by the process, is always driven down. In global optimisation methods such as simulated annealing, there is a non-zero but increasingly small probability as time goes by that the optimisation process will get worse again before further improvement, thus helping it to avoid being trapped in globally sub-optimal local minima in the early stages. In engineering, the equivalent occurs when a new technology is introduced which first of all makes things worse before making them much better. All of this is denied us in software engineering because we would not recognise a good technology if it fell on our collective heads because most of us keep no measurement trail in general and we do not analyse failure. Indeed in some companies, the word 'failure' is specifically forbidden echoing the management delusions exhibited in London's tragic King's Cross fire some years ago, where the word 'fire' was deliberately replaced by the rather less threatening 'smoulderings'.

By definition, repetitive failure means the re-occurrence of an existing mode of failure. Only 25 years ago, even mechanical engineering was still prone to this as evidenced by the DC-10 cargo door fiasco as chillingly analysed by Nancy

Leveson, (Leveson 1995). In essence, prior to the dreadful crash of the Turkish Airlines DC-10 in March 1974 after the baggage door fell off depressurising the aircraft and severing key control cables, there had been no less than 1000 baggage door incidents reported amongst a fleet of less than 100 DC-10 aircraft in the 6 months prior to the crash. In spite of repeated warnings from organisations as prestigious as the Dutch National Aviation Agency, Rijksluchtvaartdienst (RLD), nothing was done until 346 people paid with their lives. There is overwhelming evidence for the abundance of repetitive failure in software systems starting with Ed Adams' ground-breaking study at IBM in the early '80s, (Adams 1984), where amongst other things, he found that the most common failures in IBM operating systems were caused by only 2% of the faults. In other words, the repetitive failure of a small class of faults dominated the failure behaviour of the system. Meanwhile 13 years later, I found the same thing, (Hatton 1997b), demonstrating a wide class of re-occurring faults which continued to fail in both C and Fortran programs, so no change there.

At the risk of upsetting people and to put it in the bluntest possible terms, we cannot possibly call ourselves engineers while we continue to let this happen. We are simply dabblers allowing unconstrained creativity to cloud our analytic minds with dreams of what might be. The process corrupts even our programming languages where an immensely complicated language like C++, full of experimental features which we do not yet understand properly can be released onto an unsuspecting world and the next minute appear in a nuclear reactor control system because somebody got taken in by the "potential". I have actually seen this. This is Never-Never Land writ large. C++ actually started by inheriting a wide class of faults in C which are known to fail, thus paying homage to the twin gods "backwards compatibility" and "market share" and then proceeded to add new ones of its own. I do not wish to be unfair to the many very capable people who have contributed to the development of C++, but I'm sure they would be the first to agree that we do not yet understand well enough how this language fails, to justify its use in a critical system. We all demand backwards compatibility and yet this cripples any attempt to remove known points of failure. To do this, you have to start again with a new language as has happened with Java, which actually discards some of the wilder elements of C as its starting point, and this then takes the risk of not achieving enough market share to succeed. The Darwinian analogy is that only dinosaurs can inherit the earth because they sit on everything else. This is adequately evidenced by a veritable litter of languages of the recent past including Algol and Pascal which were far better specified than most modern equivalents but whose use has withered essentially for fashion reasons.

It goes on … Recently, I opened the latest edition of some computer journal or other to be greeted by the heading "Objects are dead, long live the Component". It appears as though somebody has re-invented the subroutine, first described by Alan Turing in the late '40s. Only this time it will be too complicated to work properly no doubt. These may seem rather cynical words, but I am tired of seeing the same mistakes occur again and again and I am tired of having new acronyms thrust at me every week exhorting me to change my working practices at somebody's misguided whim. I am certainly not the first person to feel this as can be seen by Tony Hoare's essay "Programming is an Engineering Discipline", (Hoare 1982). Hopefully, software failure on the scale which the year 2000 promises, will bring home to everybody how much we have lost our way, and how much we depend on an essentially flawed technology. We are quite capable of producing excellent, reliable software as the experience of the estimable Linux and a number of promising technologies amply prove but we will probably have to learn some more painful lessons first. As was quoted memorably in the BBC television program "Locomotion", broadcast on Nov. 7th., 1993, "Disasters are the motivator of progress and innovation [in engineering]".

Feedback in engineering

In my various discussions to companies in recent times, a very frequently asked question is "How do we start a measurement process ?". It is disturbingly easy to believe that nothing can be done unless a sophisticated measurement process is in place. As a result, many companies fall at the first hurdle. Even those that succeed in setting up such a process fail to analyse the data successfully. It must never be forgotten that measurement on its own is simply an overhead. Analysis of the measurements and feedback into the measured process is necessary to garner the desired effect of improvement as succinctly depicted in Figure 2. So how do you start ?

The most useful practical advice I can give is that history teaches us that *a crude measurement is sufficient to improve a crude process.* Therein lies one of my (few) criticisms of the CMM. In the CMM, measurement tends to wait for there to be a defined process, which is a comparatively high level which can take many years to achieve. However, in my experience there are many opportunities for simple measurement-based guidance even in the early chaotic days. The most important thing to realise is that measurement itself is a process which must be subjected to incremental improvement. In other words *the measurements should evolve with the process being measured.* This can be achieved in practice by continuing to ask

pertinent questions of the measurement system. If reliability is the subject of an improvement goal, measure defect data. If the data as contained cannot answer a process based enquiry about defects, change the way the data are acquired. It is usually very obvious whether the data are sufficient and I have seen a number of voluminous and expensive measurement systems which failed this simple test. For a defect measurement system for example, the very minimum it should be able to answer is “Where ?, When ?, Why ?, How severe ? and Could we have found it earlier and if so where and how ?”. Many systems in my experience miss at least one of these *sine qua non* items, an observation reflected by Shari Pfleeger in (Pfleeger 1998).

The very best feedback is achieved when analysis of failure is easy because of a direct and easily identifiable link between effect and cause. This *absolutely* depends on the subject of the next section.

Diagnostics

I deliberately inserted in the title of this essay, the *lost* art of diagnostics. If the essence of engineering improvement is the analysis of failure, then anything which facilitates this process is important. Good diagnostics are of paramount importance in understanding systems failure. In spite of this, many efforts at inserting diagnostics in software systems are risible to say the least. Indeed, many systems engineers still take them *out* after testing before releasing the final product, thus crippling the diagnostic process when an inevitable failure occurs in the field. *Repetitive failure flourishes in a diagnostic vacuum.*

Diagnosing failure in a complex, tightly coupled system can be extremely difficult. To give the reader some idea of this, take a semi-colon or some other important syntactic token out of a C program or whatever your favourite language is and watch the error cascade from the compiler. Near the source of the error, the error messages will hopefully be reasonable, (although not always as will be seen later in this section). Further down the error cascade, the messages become increasingly remote from reality. In real life, a memorable example of this is documented by Peter Mellor in (Mellor 1994) whereby an error reported on the pilot’s console of an Airbus of “MAN PITCH TRIM ONLY”, rapidly cascaded error messages around the numerous computer systems in the aircraft finishing up with the definitely very inscrutable and entirely spurious “LAVATORY SMOKE”. Now imagine that your diagnostic system *only keeps the last few messages, if any*. Where on earth would you start ? Well unfortunately, this is about all that many

systems can manage. Consider the following examples culled from various sources:

Please wait ...

This refers to the celebrated message produced by both the pilot and co-pilot's consoles in Airbus A340 G-VAEL in September 1994, (AAIB 1995). A number of other interesting software defects arose at the same time. A more appropriate message to the pilots of nearly 300 people on a final approach to London's Heathrow airport might have been, "The Flight Management System has crashed. It is now being rebooted, a process which will take slightly less than N of your Earth minutes". I would be delighted to stand corrected, but as far as I am aware, the fault or faults which lead to this failure have still not been tracked down and it is known to have occurred in several other aircraft of this type and is, and hopefully by now was, therefore a repetitive failure mode.

System stressed ...

This serious incident occurred in my local bar into which I had gone for a drink with an old friend and fellow computer enthusiast. On approaching the bar, the manager said "I'm sorry, I cannot serve you, the cash registers have crashed and there is a funny message on them". The message was "System stressed …". Before any unpleasant crowd scenes could occur, (being unable to buy beer is one of the few things the English get worked up about), my friend and myself volunteered to help and immediately started fiddling with the system talking excitedly about full stacks, communication deadlocks, protocols and other learned topics. After some time and absolutely no progress, the manager's ghostly voice emerged from the depths of the cellar saying, "Shouldn't the printer have some paper in it ?". "Er, yes, it would help", we answered. The reason why the "System stressed …" message had appeared now became dreadfully clear. The printer had run out of paper, the print queue had filled up and the system had stopped to wait. The idea of putting the message "Printer out of paper" or the terser but equally acceptable "No paper" on the system console had obviously not occurred to the literary giant and author of this widely used cash management system. Somewhat chastened, my friend and myself immediately promised never to call ourselves knowledgeable about computers again. (I also know how Peter Mellor felt when aided by some particularly illiterate documentation, he managed to erase the system disk of the charity organisation he had volunteered to help immediately before his adoption of a foetal position in the kneehole of the nearest desk, (Mellor 1994)).

More than 64 TCP/IP or UDP messages ...

This is an example of how a little knowledge can be a dangerous thing. I know what TCP/IP and UDP are and have written rudimentary systems based on them over the years. When this appeared recently on my brand-new G3 Macintosh with the latest operating system incarnation OS8.1 when trying to dial my Internet Service Provider for the first time, I immediately turned to my books and 'years of experience' and started musing about what on earth it could possibly mean. I fiddled and tinkered with internet addresses, POP accounts, mailboxes and all manner of things before I realised two desolate hours later that the external modem wasn't switched on. I would submit with all due respect to the Apple Corporation and its fine machine, that a slightly more appropriate message for diagnostic purposes might then be "Modem not responding".

Line 27: Incompatible types ...

A personal favourite of mine, this is an example of a message that one of my compilers occasionally produces. This one occurred in a C program, for which the ISO C90 standard in a splendidly arcane piece of prose, defines type compatibility more or less as types which are the same but not identical. You may need to think about this for a little while ... The normal reaction of all C programmers I have taught who come across this is to remove statements desperately until the offending message disappears without ever knowing why it appeared in the first place, so it re-occurs. The irritating thing of course is that the compiler writer knows precisely what the types are and where they are defined. The message could have said, "line 27: you have re-declared a type as 'signed char' when it was previously declared as 'char' at line N, These are incompatible, see section N.N.N.N of the ISO standard". The message could have been added in a few minutes by the compiler writer to save countless hours of panic amongst C programmers around the globe, simultaneously educating them so it doesn't happen again, but no, that would be giving too much away.

Conclusions

I could go on in this relatively light vein as I see many, many examples of fundamentally poor diagnosis crippling the ability to analyse a failure and find out why it happened. In fact, it is not uncommon to find fully a third of all failures in a system unattributable to any fault or faults, and as a consequence, they continue to occur. These are not momentous intellectual feats of computer science and perhaps that is why they are neglected so frequently, but to ignore the fundamental position

of good diagnosis in a complex, tightly-coupled system is to throw away all hope of ever understanding failure and thereby preventing its future repetition.

The true essence of good engineering is the recognition that all systems fail. It is incumbent amongst all software engineers that they recognise this in several ways:-

- by minimising the number of injected faults within the available budget by using appropriate measurement and feedback techniques. They exist as can be seen by reading the other papers in this special issue
- by designing the system to fail gracefully
- by designing the system to facilitate diagnosis so that failure can be easily related to the responsible fault or faults to make sure that repetitive failure is systematically eliminated from the system

This requires better training and much more attention to the measurement strategy, testing strategy and failure recovery procedures but Glen Myers was saying this twenty years ago, which is where I came in.

I fervently hope that we are not still having to repeat this message in another twenty years.

References

AAIB (1995). AAIB Bulletin 3/95, Air Accident Investigation Branch, DRA Farnborough, U.K.

Adams, N. E. (1984). "Optimizing preventive service of software products." IBM Journal Research and Development **28**(1): 2-14.

Dawkins, R. (1996). Climbing Mount Improbable. London, Penguin Books Ltd.

Fitzpatrick, S., C. D. Gelatt Jr, et al. (1983). "Optimisation by Simulated Annealing." Science **220**(13 May 1983): p. 671-680.

Hatton, L. (1997a). "Software failures - follies and fallacies." IEE Review **43**(2): p. 49-54.

Hatton, L. (1997b). “The T experiments: errors in scientific software.” IEEE Computational Science & Engineering **4**(2): 27-38.

Hatton, L. (1998). “Does OO sync with how we think ?” IEEE Software **15**(3) May/June 1998: p. 46-54.

Hoare, C. A. R. (1982). Programming is an Engineering Profession, Oxford University Programming Research Group.

Leveson, N. G. (1995). Safeware: System Safety and Computers. Reading, Mass, Addison-Wesley.

Mellor, P. (1994). CAD: Computer-Aided Disaster, Centre for Software Reliability, City University, London.

Pfleeger, S. L. (1998). Measurement and Testing: Doing More with Less. ICTCS'98, Washington.