Laboratory Computer Software: Fitness for purpose

Les Hatton
CISM, University of Kingston*

June 12, 2008

Abstract

This short paper summarises the impact of 50 years of computing research on the successful implementation of computer software which, in essence, is software which does what is required of it most of the time. It discusses known problem areas under a series of headings and makes recommendations for practitioners to achieve this goal.

1 Overview and some obligations

It will not come as a surprise to anybody trying to write computer software of any significant size that it is not easy. Indeed 50 years of computing research suggest that in the average programmer's career, the chance of them producing a significant computer programme without defect is effectively zero. If they were by chance to produce one, they would:-

- Not know it
- Not be able to prove it
- Not be able to repeat it systematically

This provides an appropriately humble starting point. It is therefore an unfortunate truth that for any reasonable level of complexity, computer software is very likely to contain defects *which are essentially unquantifiable*. This in turn forces two obligations on the producers, particularly an a discipline where failure can be hazardous:-

- Any failure in the system must have the least possible impact on the end-user
- It must be possible to find out why the system has failed so that it's cause can be quickly and accurately corrected.

These two obligations are at the heart of good engineering and both are *design* issues. It is vital that software producers think about both the functional and dysfunctional behaviour at the beginning. Dysfunctional behaviour is the behaviour exhibited by a system given the wrong inputs for example.

*L.Hatton@kingston.ac.uk

1.1 Expected reliability and how good is good ?

In a discipline as bereft of the scientific method as computing, it has been historically difficult until recently to define even a scale of quality in terms of acceptable failure. Today however, we are beginning to have some idea. If we define a fault as a static property of a computer program, for example, the use of an uninitialised variable or a divide by zero, (there is a very wide class of such faults), then a computer program is good if it exhibits less than 1 fault that fails in the entire life-cycle of the code for each 1000 lines of executable code, (KXLOC). Executable code is code which the compiler or interpreter will execute in some way.

On this scale, the very best programs ever produced, (this includes the NASA Space Shuttle software and the Linux kernel for example), are around about 0.1 / KXLOC. There are also examples in the literature of commercially produced systems in the range 0.2-0.4 / KXLOC. Systems down at the 0.1-0.2 range are observed to fail very rarely.

Systems as good as this give much cause for optimism. All we as a community have to do is to find out why systems get to be this good. The rest of the article will attempt to answer this question.

1.2 The cost of failure

The cost of failure is so large that it beggars belief. We are perhaps all familiar with the enormous cost of failed systems in the NHS but it is fairly widespread. Prestigious studies such as that carried out in the USA by the National Institute of Standards and Technology, [11] and in the UK by the Royal Academy of Engineering [12], highlight an endemic problem of the order of billions of pounds per year in the UK alone, and one which will not simply go away.

2 Technology and individual competence

Perhaps the biggest surprise in an industry awash with technology is that it doesn't seem to make much difference. By far the biggest factor which emerges in most studies is the individual quality of the engineers who build a system and a few common-sense principles. We have known this since the admirable book by Fred Brooks, [6] which every aspiring software producer should read. Rather than absorbing this powerful lesson, the computing industry became obsessed with the notion that the process or bureaucracy of building software was the most important part. The evidence is to the contrary. No matter how well-defined a process, the quality of the product still depends mostly on the quality of the engineers who build it. To this, a good process can bring consistency and accountability.

This whole area is exacerbated by the currently falling rolls in computer science, an enduring trend since about 2001 throughout Europe, the USA and Australia, where rolls have dropped by around 50% in this period.

Train your engineers and never stop training them. Pay them well enough that they don't need to seek management roles to improve their salary and try to encourage them to stay with you a long time. Engineers with 10 or more years of demonstrably skilled development are of inestimable value.

2.1 The underlying architecture and open source

This certainly has a part to play. It is of little use producing a high quality application if the architecture on which it runs is not of comparable quality. Linux-based systems appear to be reaching very high levels of reliability, (mean time between kernel failures of the order of hundreds of years is being reported by some cluster users). This is typically rather higher than some commercial alternatives. It is not uncommon to find some companies afraid of open source even though they unconsciously depend on it already to a very large degree, (a very significant amount of web infrastructure is built on open source).

2.1.1 Spreadsheets

Spreadsheets should be used very cautiously. *It is common for users to store data in them but this should never be done.* Data should be stored in databases for the simple reason that extracting data from multiple spreadsheets of arbitrary structure is effectively impossible and data quality control is often poor.

2.1.2 Open document formats

Never underestimate the importance of open formats. Proprietary document formats should never be used because they force dependence on particular suppliers and even if a supplier stays in business, it is not uncommon for later versions of a piece of proprietary software to be incapable of reading older versions, (because the software investment to do this is high), so legacy documents can become effectively unreadable within just a few years.

3 Some successful strategies

It is worthwhile highlighting a few practical development tips which are frequently observed in high quality systems. I will not try to associate any particular "buzz" words with these as many technologies claim to have invented what is often common-sense.

- **Choice of a programming language**. Much effort has been expended on inventing new languages by the computer industry over the years and even today, language choice remains an emotive issue. It shouldn't. All the available evidence suggests that it doesn't appear to make much difference, [2]. However, it is certainly true that a programmer needs to be experienced in that language.

  Do not switch programming languages unless you have to. Know the language you use. Don't experiment with it on real projects

and write it as you would like to read it. If possible, use a language for which there exists an international standard, (for example, C, C++, Ada and Fortran) rather than one that doesn't, (for example Java). This way, you can validate the compiler if you have to. Finally, simplest is always best.

- **Peer review and repeatability**. Design and code inspections by independent persons are amongst the most effective way of eliminating defects that have ever been discovered, [1], [5], [13]. In spite of this, they are still poorly implemented and hard-pressed software managers find it hard to believe in them. This is one of the most significant factors in the quality of Open Source projects such as Apache and the Linux kernel itself.

  One of the biggest mistakes in code inspecting is to go too fast, (to keep costs down). Most researchers in this area argue that this should be no more than 100 lines of code an hour and preferably less, [1]. At this speed, inspections can be astonishingly effective.

- **Prototyping**. It is relatively rare for the production of a significant software-controlled system to unfold in a trouble-free and entirely predictable manner as the so-called *waterfall* model of software development invites us to believe. In most systems, the only point of comparison with a waterfall is that when you fall off the top, there isn't much you can do about it. Instead, software development is normally defined by iteration. Grand top-down designs such as have been attempted many times in Government IT projects normally fail in grand top-down ways. In contrast, the most successful systems build incrementally by iterating from simple ideas to larger-scale implementations.

  Never stop prototyping to make sure that you and your intended users know where you are going.

- **Testing**. Most developers have no idea how to test a system even though testing is a well-understood and well-researched area. Simply handing a system over to a specialised testing group is not the answer as this delays defect discovery, which in turn increases expense. The NIST study cited above, [11], quoted that far too much emphasis was placed on finding defects later in the development cycle.

  Train developers in testing. Testing is not a low-level activity that anybody can do. It is a highly sophisticated and effective procedure when done properly.

- **Code re-writing**. Although strictly speaking, this could be included under prototyping, in practice it is worth stating separately that programmers too often stick with their first software version. It has been known since the early experiments of [8], that re-writing the same code several times leads to very significant improvements in readability and reliability.

  Re-write your code until it reads well and you are happy with it. This usually takes three attempts but may be more on difficult algorithms.

- **Scientific coding**. Programmers should never be satisfied that because a result looks right that it must be right. Subtle but significant defects can remain in code for years without being spotted, corrupting the output of programs perhaps irretrievably, [4], [3]. This is very difficult to control and the most effective way of finding such defects seems to be by comparison of different software packages written to the same specifications independently. The failure modes aren't independent even then but as a diagnostic tool, it appears very effective.

  > *Don't be satisfied when your output seems right. Instead design tests to find defects right down to the level of precision at which your results need to be accurate.*

- **Little and often**. Build programs little by little. In a complex system, tracing injected faults back through many changes is often infeasible. Modern machines are very fast allowing the developer to build a little and then test. The technique is known as incremental building and in practice is very effective when accompaned by good automated test suites. Incremental building should also be done using revision control and build systems, (for example *rcs* and *make* on Linux), to record each increment so that back-tracking becomes simple.

  > *Developers should be able to rebuild a system including the tests relevant at the time, at any time slice back to the beginning of testing, particularly if the cost of failure is high. There is simply no excuse for not being able to do this.*

- **Run-time checks**. Programmers often use these during testing and then take them out before delivery, (when the software is in the state known as 'tested'). If the system then fails at run-time, it is often impossible to diagnose why. In addition, the work of [14] shows that such checks are just as effective in a tested system.

  > *Developers should never remove diagnostic code from released programs, although any diagnostic output will need to be stored or communicated.*

4 Quality of Interfaces

The quality of many modern interfaces to computer systems is so bad that it is worth a section in itself. To illustrate, consider the following examples, some of which are safety-critical and others merely irritating:-

- In 2008, the "handbrake" of a Volvo was relegated to a push-button with an indicating light hidden at knee level. It is perfectly possible to engage this without realising it, (I did). Other car manufacturers have copied this and it is now impossible to stop some makes of car without going through software. Given that there have been numerous well-publicised failures in electronic software controlled systems in cars, [10], this is a highly questionable pursuit.

- In 2007, the main console control system in a BMW 3XX series consisted of a multi-functional joystick controlling a nested menu system on a screen in the middle of the driver's console. Even simple and common activities such as turning the radio off were controlled through this interface. Unlike the manual systems it replaces, using it necessitates taking your eyes off the road for significant periods of time to operate the menu system.
- In 2005, a trader entered a trade for 600,000 shares at 1 yen each rather than 1 share at 600,000 yen losing Mizuho Securities in Japan some 60 billion yen, [7]. This is so common in financial circles, that it is known as "fat finger syndrome". In reality, much of this problem should be ascribed to the programmers who failed to do any kind of reasonable field checking.
- On a Sony-Ericsson Cyber-shot (and many other mobile phones), the "Delete Message" and "Delete all messages" choices are next to each other making it painfully simple to select all rather than one on a keypad which is far from easy to use.
- Non-intuitive behaviour in aircraft systems is surprisingly common, [9] for example.
- One of the prime ways of breaking systems is to feed them incorrect input data as programmers rarely check fields properly, [15].

  The interface presented to the end user needs very careful thinking about and there is abundant evidence to show that it is often poorly thought out. The designer should never assume that the user will enter sensible data. All fields should be fastidiously checked for reasonableness with responses the user can understand for those data which are not reasonable.

References

[1] T. Gilb and D. Graham. *Software Inspections.* Addison-Wesley, 1993. ISBN 0-201-63181-4.

[2] L. Hatton. Software failures, follies and fallacies. *IEE Review*, 43(2):49–54, 1997.

[3] L. Hatton. The t experiments: Errors in scientific software. *IEEE Computational Science and Engineering*, 4(2):27–38, April 1997.

[4] L. Hatton and A. Roberts. How accurate is scientific software ? *IEEE Transactions on Software Engineering*, 20(10), 1994.

[5] W. Humphrey. *A discipline of software engineering.* Addison-Wesley, 1995. ISBN 0-201-54610-8.

[6] F.P. Brooks Jnr. *The Mythical Man Month.* Addison-Wesley, 1975. ISBN 0-201-00650-2.

[7] Leo Lewis. Fat fingered typing costs a trader's bosses GBP 128million, December 2005. http://www.timesonline.co.uk/tol/news/world/asia/article755598.ece.

[8] M.D. McIlroy and J.W. Hunt. An algorithm for differential file comparison, 1976. Bell Labs CS Tech. Report 41.

[9] Peter Mellor. CAD: Computer-aided disaster. *High Integrity Systems*, 1(2):101–156, 1994.

[10] Tim Moran. What's bugging the high-tech car ?, Feb 2005. New York Times, 6th Feb, automobiles section.

[11] Michael Newman. Software errors cost US economy $59.5 billion annually, June 2002. http://www.nist.gov/public_affairs/releases/n02-10.htm.

[12] Royal Academy of Engineering. The challenge of complex it projects, 2004. Royal Academy of Engineering report, London, ISBN 1-903496-15-2.

[13] S.L. Pfleeger, L. Hatton, and C. Howell. *Solid Software*. Prentice-Hall, 2002. ISBN 0-13-091298-0.

[14] M.J.P. van der Meulen and M.A. Revilla. The effectiveness of software diversity in a large population of programs. *IEEE Transactions on Software Engineering*, 2008.

[15] James A Whittaker. *How to Break Software*. Addison-Wesley Longman, 2002. ISBN 0201796198.