"Programming technology, reliability, safety and measurement"

by Les Hatton

Oakwood Computing, UK

lesh@oakcomp.demon.co.uk; http://www.oakcomp.demon.co.uk/

Abstract

In this short paper, we discuss the influence of programming technology on reliability and on safety, and why the industry seems unable to learn from its mistakes, apparently preferring to chase pots of gold with OO and such like written on the side.

Discussion

Safety and reliability in engineering are end-to-end issues and the strength of the whole chain of development is the strength of the weakest link. In a software system, the safety and reliability of the whole system depends on the interaction of myriad pieces of hardware and software in an often highly unpredictable way. A single software mistake can be entirely benign or absolutely catastrophic depending on its interaction with the rest of the system.

Consider the Ariane 5 crash in 1996 for example. An analysis of the chain of events which led to this teaches us much of systems engineering. In Ariane 5, according to the author's interpretation, the following chain appears to have led to the crash, (Lions 1996):-

1. A 64 bit floating point number was coerced into a 16 bit integer in a non-critical component. That this should be necessary is a highly questionable feature about the original design. (The software was written in Ada, an unusually secure language for type changes and the programmer has to deliberately override the compiler's complaints to achieve this, unlike other less strict languages such as Fortran, C or C++).

2. Very shortly after the launch, this coercion overflowed. In Ada, this generated an exception which in this case was allowed to lead to a system shutdown.

3. The shutdown, not only shut down the non-critical components, it shut down critical components running concurrently.

4. Ariane 5 employed dual channel diversity, but the same software was running in each channel, so both channels shut down within a very short interval of each other. This proved fatal and Ariane 5 was then doomed.

Interestingly enough, if a sloppier language such as Fortran, C or C++ had been used, the coercion would have overflowed silently, the non-critical component would have continued running incorrectly but irrelevantly and the crash would likely not have occurred. So here, arguably the most secure language ever designed led to a fatal crash because of the failure of its programmers to understand the full implications of its detailed exception handling, where use of a sloppier language would have not. Other design decisions also intrude. Had a true diverse system (with independently derived software in each channel) been used, the crash would have been much less likely to have occurred.

The point of this argument is to show that programming technology, in particular, the choice of language, is just one of the issues which contributes towards the safety and reliability of a system in which it is embedded and cannot be considered in isolation. In fact, it seems that system reliability when measured in defects per KLOC (thousand lines of code) is relatively independent of programming language, (Hatton 1995), or at least, there is no clear relationship.

To make the point another way, consider two very different software application areas such as a word-processor and a medical drug injection system. To a word-processor user, reliability is not really important compared with the features and the user will typically tolerate regular failures providing there is an auto-save option every few minutes so that not too much work is lost. In a medical drug injection system however, reliability is of absolutely paramount importance. Any unreliability has a significant chance of killing the patient attached to it. Such issues as price and performance pale into insignificance by comparison. *However, the applications in both cases might very well be written in the same programming language and yet achieve entirely different levels of reliability.*

As an example of this consider Table 1, which shows the author's experiences with certain widely available software systems. Most of the individual packages which contributed to these results have been written in C or C++ and yet the stark differences in reliability show that this

can have little to do with the programming language. There are obviously wider issues afoot here including programmer fluency in the language, quality of testing and so on.

Environment	Observed reliability
Windows'95 + Professional Office	1 defect every 42 minutes; 28% reboots
Macintosh OS + Microsoft Office	1 defect every 188 minutes; 56% reboots
Various flavours of Unix	< 1 per year; no reboots
Linux	None yet recorded in 3 months of medium load.

Table 1: Personal experience with various machines in the last few years. The Window's 95 experience is based on around 6 months use but the defect rate slowly got worse through this period.

(It is interesting to contemplate the Unix v Windows 'wars' raging throughout corporations around the world in the light of this table. Corporations obviously have scant regard for reliability even though network unreliability is arguably one of the most expensive forms of unreliability).

Given the end-to-end nature of reliability and safety, in order to discover just where the weakest link of software systems lies, measurement is absolutely paramount, however this has severe problems for software systems as painstakingly pointed out by (Fenton 1991) and (Fenton, Neil et al. 1995) for example. The result is that software engineering is currently far closer to the fashion industry than to any engineering discipline.

The measurement of reliability

How can reliability be measured ? To understand some of the difficulties, consider the meaning of the four commonly used words to describe misbehaviour. These are as follows:-

failure

When the software system exhibits unusual behaviour when running. It is important to note that any misbehaviour is a failure, even when it does not prejudice the run-time behaviour of the system.

fault

This is normally used to represent a potential failure. In other words, under the right circumstances, a fault will lead to a failure. Unlike

failures, faults can be identified without running the software, for example, using inspections. However, even a fault guaranteed to cause failure may not fail in practice, for example if the particular piece of code containing the fault is never executed. In contrast other faults can fail repeatedly. Some organisations only measure serious fault, which occurs some 10-20 times less often, and such use will lead to very misleading measures.

error

A problematic term used in different ways in different standards. In the IEEE model and others, we have:-

error -> fault -> failure

Whereas in IEC 1508 for example, we have

fault -> error -> failure

Here we will not use the term at all.

defect

Frequently used as a generic term for fault or failure, depending on the circumstances.

Some measures of failure rate

Having clarified the terms used, note also that their rate of occurrence depends on the nature of the product, although obvious measures generally present themselves in each case. The two most important measures are:-

a) Mean time between failures, (MTBF). This would be a typical measurement for a continuously running piece of software, such as a flight management system in an aircraft. Precisely the same measure is used to assess the reliability of a hard-disc drive with quoted MTBF values of 500,000 hours nowadays, (far more than would be expected of a software system). In general, this measure is not often used with software owing to the practical difficulties of measuring system time since the last failure.

b) Probability of failure on demand. This would be a typical measurement for a nuclear reactor control system, for which the software only begins to run when an unusual event occurs and it is

the probability that it will not perform to its specification when started which is of interest.

By far the most common measure of reliability is the *defect density*, because of its essential simplicity. In essence, it is a measure of how many defects have been found per 1000 lines of code (KLOC) at some point in the life-cycle. It is important to realise that this is a function of usage time. Figure 1 illustrates the typical behaviour of defect density when it is used to measure failure rates after a software system has been delivered.

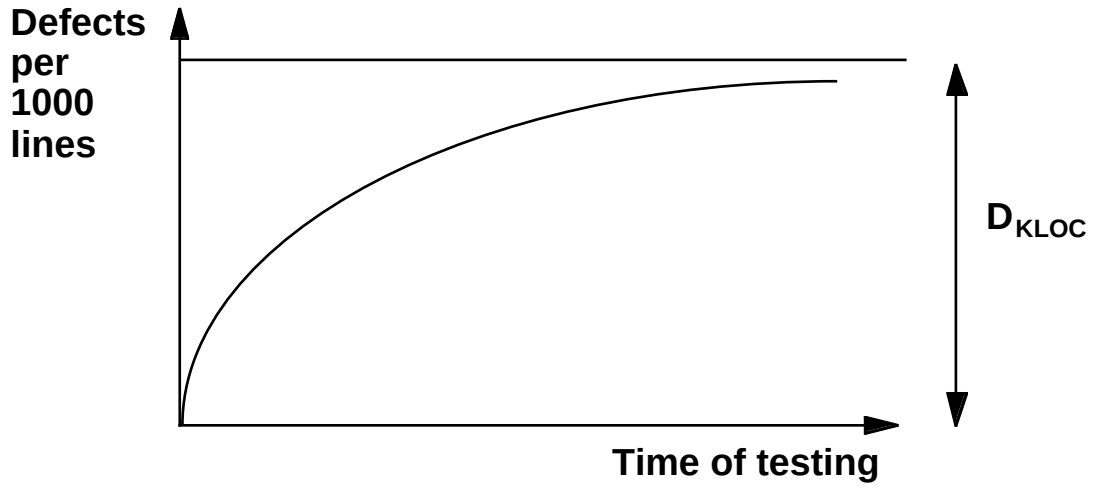

Figure 1: Illustrates the behaviour of the defect density curve as testing time increases. To begin with, there are no defects. As the product is used, defects are found and fixed and eventually, the defect density tends to asymptote to some value. It only makes sense to compare asymptotic values.

Typical defect density measures

There seems to be reasonably wide agreement that a good quality commercial product will have an asymptotic defect density in the range 3-6 per KLOC, whereas very few systems stay below 1 defect per KLOC throughout the life-cycle when all defects are counted, (Hatton 1997), (Musa, Iannino. et al. 1987). Although defect density is not directly related to measures of failure rate, a "1 defect per KLOC" system tends to be very reliable indeed.

Progress by incremental improvement

Study of many software failures suggests that a very significant fraction are repetitive in nature with type mismatches as occurred in Ariane 5 for example repeatedly implicated. In other words, the industry continues to repeat its mistakes. The first sign of a maturing industry is the recognition of this and the elimination of known failure modes.

Unfortunately, software engineering is not there yet and continues to commit them. As an example of this malaise, Figures 2 and 3, taken from (Hatton 1997), show the statically detectable fault rates of a wide class of problems in C and Fortran respectively which are known to mature into failures. These known failure modes continue to occur even in recently written software. They can all be detected automatically but in many software processes such detection methods are simply absent. In other words, none should be present in a released system as they are preventable by techniques the industry already knows to do.

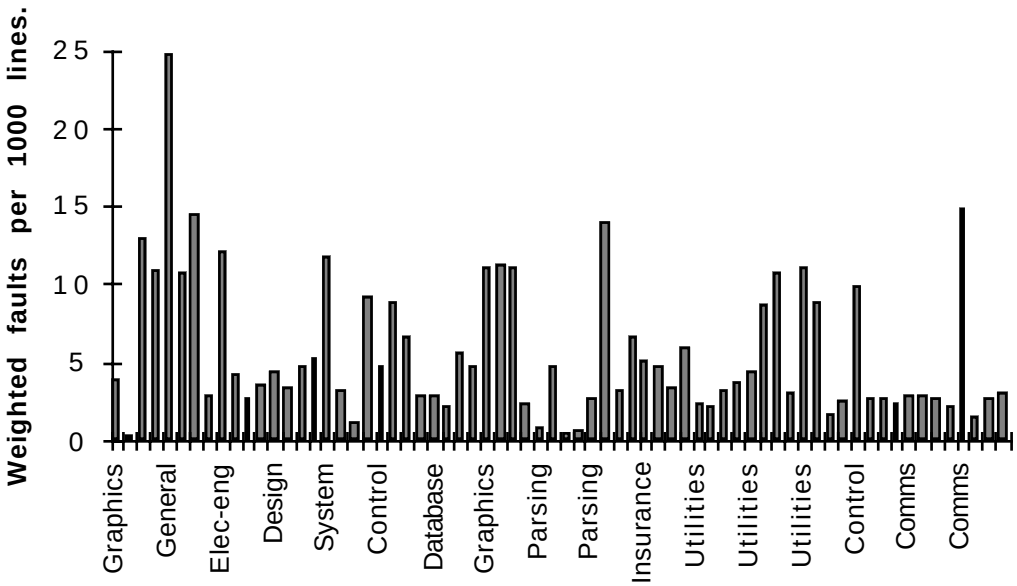

Figure 2: Weighted fault rates per 1000 lines of code for a wide variety of commercially released C applications plotted as a function of industry.

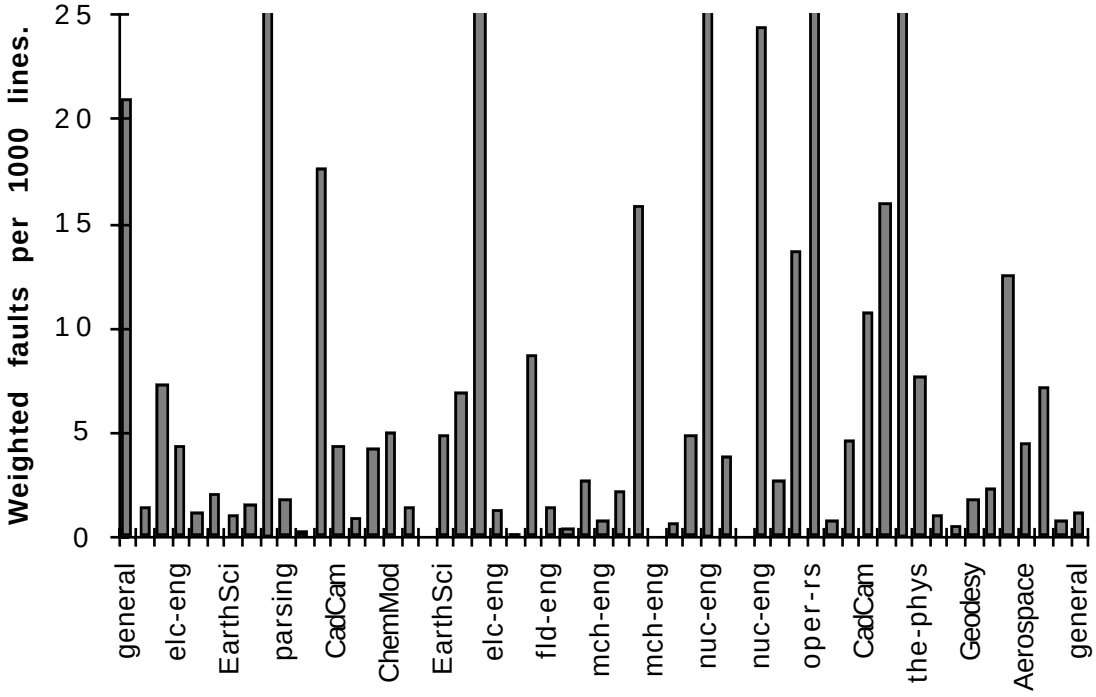

Figure 3: Weighted fault rates per 1000 lines of code for a wide variety of commercially released Fortran 77 applications plotted as a function of industry. The five tallest all go off the top of the graph.

From this and other sources, it would appear that a significant amount, perhaps 40%, of all known software failures could have been prevented by techniques the industry already knows how to do. Of course, it is clear from the figures in Table 1, that the industry can produce exceptionally reliable code if it wants to with Linux, (whose source code distribution guarantees the effectiveness of inspections amongst other things), a shining example. However, instead of consolidating on the technologies which can provide enhanced reliability such as inspections, the industry appears to wish to pursue new and absolutely untested paradigms instead.

Progress by guesswork - OO and C++

To illustrate the dangers of changing paradigms without measurement, this paper will finish with a discussion of OO based systems in general and C++ implementations in particular.

In the last 10 years, much of the thrust of software development research has been into Object-Oriented (OO) methods. This has taken place more or less in the complete absence of measurement and yet the paradigm is so firmly established that, if it is found to be wrong, an enormous amount of money will have been invested on no evidence

other than its ‘intuitive appeal’. This is clearly not good enough given the many surprises software engineering holds.

There is considerable data from sources such as (Boehm 1981), (Arnold 1983) to suggest that corrective maintenance consumes around 40% (50% of 80%) of all life-cycle costs. So any paradigm desiring to be considered a step forward should address this problem. However, in spite of the amount of time and money lavished on it, recent data shows that C++ implemented OO systems actually appear to *increase* this cost significantly with no attendant benefit as yet.

Figure 4 shows the costs of correcting defects in C++ based OO systems at different stages compared with Pascal non OO based systems as published by (Humphrey 1995). As can be seen, the aggregate increase is significant across the various categories, amounting to around a factor of 3.

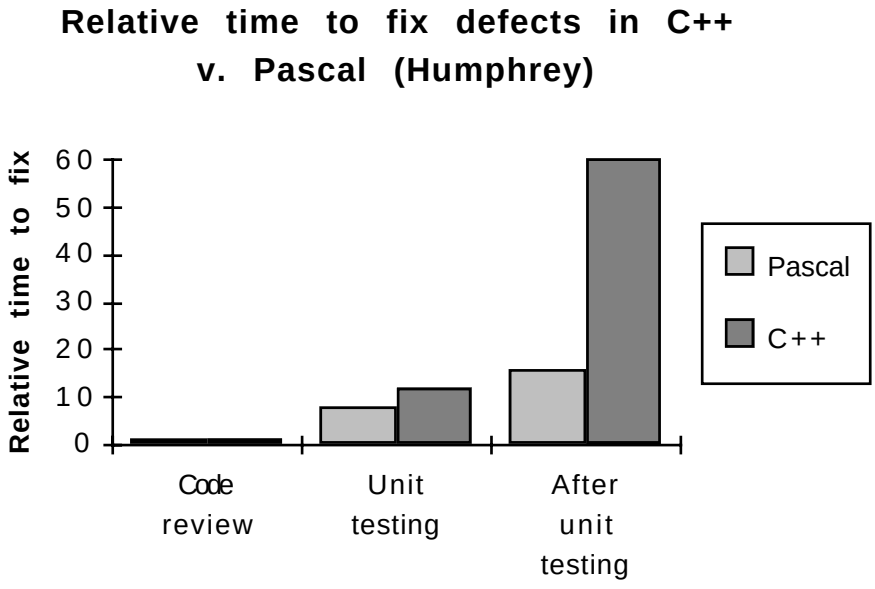

Figure 4: The average cost of correcting a defect in C++ compared with Pascal systems published by (Humphrey 1995).

Shortly after the above work was published, the author carried out a similar project comparing C++ OO based systems with C non OO based systems, the results of which can be seen in Figure 5.

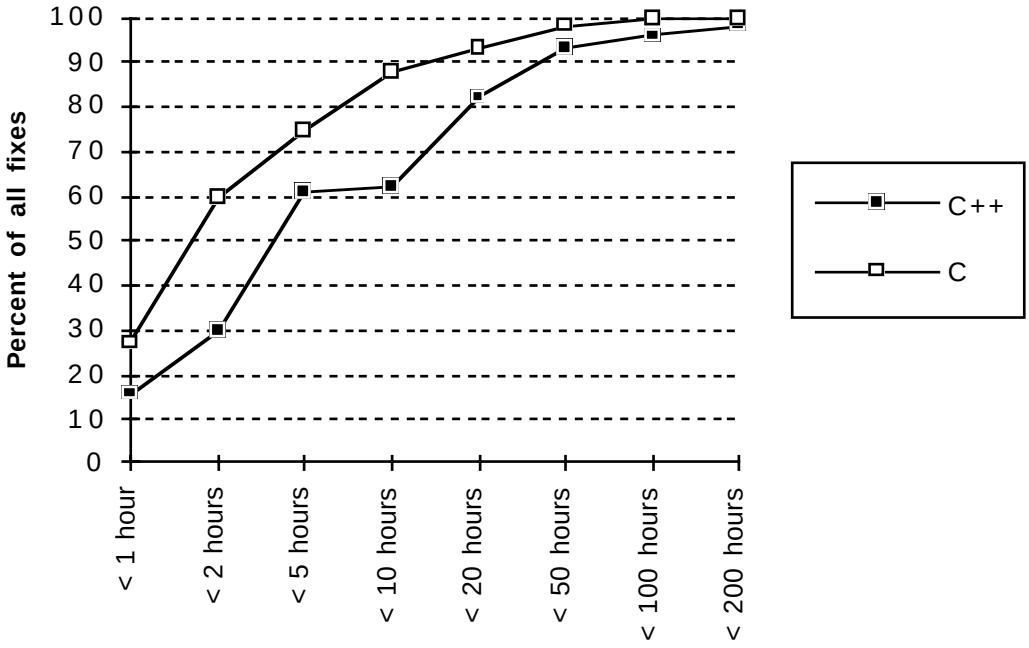

Figure 5 shows the percentage of all fixes corrected in different amounts of time comparing a conventional C system and a comparable OO C++ system. The simplest defects (fixed in < 1 hour) appear at the left. 60% of all C defects took less than 2 hours, whereas only 30% of all C++ defects were fixed this quickly. Note that around 2% of all C++ defects took over 200 hours to correct. No C defect took more than 100 hours, at which point, 5% of C++ defects remained outstanding.

Figure 5 shows the percentage of defects fixed in two highly comparable medium-sized C and C++ systems as a function of time taken to fix. As can be seen, C defects were considerably easier to fix right across the scale from simple to difficult. Once again, the overall increase in corrective maintenance cost is around a factor of 3.

This would not of course be a problem if there were far less defects, but in this project, the evidence suggests that defect densities are comparable as are the total number of defects for a particular piece of functionality. Is this then a step forward ? Well, this evidence suggests that C++ OO based systems are no more than an expensive diversion in terms of product quality when the key area of reliability is the criterion used.

Does this mean that all OO systems are damned ? What about those written in Smalltalk, Eiffel or other OO languages ? Unfortunately, there appears to be almost no data for these - truly a measurement free paradigm. However, one of the cornerstones of OO is the notion of inheritance, and in an important study recently, (Cartwright and Shepperd 1997) found that in an OO system, the defect densities in components involved in inheritance was no less than 6 times higher than in components not involved in inheritance, which is certainly food for

thought. Whether this is a natural property of OO systems is unknown at this point.

Conclusions

Programming technology and in particular, the choice of programming language, appear to have no simple link to system reliability and safety. Instead, all the available data presents a very confusing picture of an exceptionally difficult applications area and merely emphasises that without measurement there is no simple way to identify the weakest link in such systems, relegating most discussions to mere opinion. Progress is therefore likely to be haphazard at best until this situation can be rectified.

Finally, such data as there is strongly favours classic incremental engineering improvement based around the avoidance of widely-known repetitive failure modes than of the "big-bang" high-risk technologies such as OO, at least at its current state of understanding.

References

Arnold, R. S. (1983). On the Generation and Use of Quantitative Criteria for Assessing Software Maintenance Quality, University of Maryland.

Boehm, B. W. (1981). Software Engineering Economics. Englewood Cliffs, New Jersey, Prentice Hall.

Cartwright, M. and M. Shepperd (1997). An Empirical Investigation of an Object-Oriented Software System, Dept of Computing, Bournemouth University, U.K.

Fenton, N. E. (1991). Software Metrics: A Rigorous Approach, Chapman and Hall.

Fenton, N. E., M. Neil, et al. (1995). Metrics and models for predicting software defects, Centre for Software Reliability.

Hatton, L. (1995). Computer programming languages and safety-related systems. Proceedings of 3rd. Safety-Critical Systems Symposium. F. Redmill and T. Anderson, Springer-Verlag.

Hatton, L. (1997). "Re-examining the fault density - component size connection." IEEE Software **14(2)**(March/April 1997): p. 89-97.

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

Humphrey, W. S. (1995). A discipline of software engineering, Addison-Wesley.

Lions, J.-L. (1996). Ariane 5 failure - full report, European Commission.

Musa, J., Iannino., et al. (1987). Software Reliability: Measurement, Prediction, Application., McGraw-Hill.