"SOFTWARE FAILURE: FOLLIES AND FALLACIES"

Software failure is becoming big news. It costs a fortune and much of it could have been avoided. Les Hatton discusses some common misconceptions.

Version 14/10/96

Introduction and some follies

Software failure is becoming a serious issue, [1]. Ariane 5 provided a recent spectacular example of how a simple mistake, entirely avoidable, was allowed to sneak through the software verification stage and cause an immensely expensive failure. However, it is not just the aerospace industry which suffers such traumas. In January 1990, a single misplaced statement in an AT&T switching system inserted as part of a three line software "fix", caused the entire eastern seaboard of the United States to lose its telephones for several hours. The cost was widely documented at around the $1 billion mark. Again, the problem was entirely avoidable. In fact, many of the software failures we are beginning to experience share two things in common:

a) they are immensely expensive mistakes

b) they could have been avoided using simple techniques we already know how to do.

One of the reasons why software failure is becoming so pervasive is that although the public may not realise it, the amount of software in consumer electronic devices is doubling around every 18 months. The new generation of line-scan televisions contain very large amounts of code; so do cars. Even the humble electric razor contains software, euphemistically termed "head-adjusting" software, although it is not clear which head is being referred to here.

The public at large has been schooled to accept poor quality software through the medium of the ubiquitous PC. To give some example of the failure rates of such devices, Table 1 shows my own personal experience clocked up over a period of several years. Each time my own computer fails, I log the reason and effect in a spreadsheet. It might seem a little tedious, but only through measurement can we recognise a problem and learn how to avoid it. These results do not make pleasant reading and throw into stark relief the Windows v. Unix "wars" raging in corporations around the world. The one thing people do not seem to consider is reliability, so for those big corporations out there, please study this table carefully. It is a personal view only, but it is real data and is consistent with many people's anecdotal experience.

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: My 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 is getting worse.

The question is: will the public at large put up with PC levels of software reliability in a consumer electronics product ? Picture the scenario - it is the European Cup Final 1999, and the television needs to be rebooted twice during the match. If you, the reader, think this is an excessively pessimistic scenario, just wait another year for the millenium and watch what happens to the world's financial and other systems.

The main problem is that software simply is not getting much better. It would be nice to think that modern workstations, new paradigms, improvements in education, better system software and other advances have led to significant reductions in defects, and that software is therefore improving. However, in spite of repeatedly switching technologies, the number of defects per 1000 lines of source code (a common measure of software quality), is almost unchanged in the last 15 years or so. For example, Figure 1, produced by the University of Maryland's Software Engineering Lab when analysing NASA-supplied data, shows the improvement in defect density at one of the best-resourced software development environments in the world.

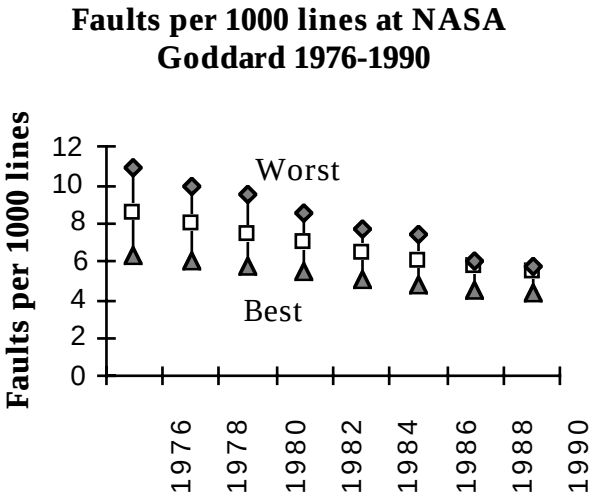

Figure 1. Graph published in the December 1991 special issue of Business Week showing the drop in software errors at NASA Goddard. If you study this graph carefully, you will see that due to an apparent cut and paste bug in Microsoft Word I hadn't previously come across, the years along the x-axis are all shifted right by one unit. I have left this in to show you just how common (and irritating) this sort of bungling is.

Note that in spite of the Trojan efforts at NASA, great concern over quality and generally excellent resources at one of the best software engineering sites in the world, *most of the improvement has been achieved by improving the bad ones, not the good ones*. The underlying improvement is only from 6 per 1000 lines to around 5 per 1000 lines in 15 years. So we are more consistent in that the difference between the best and worst in any generation has reduced by about a factor of four, but we are not much better.

Given that the amount of software is doubling every 18 months with a constant defect density, this leads inevitably to the conclusion that the number of defects will double every 18 months.

The fallacies

Why is the industry in such a mess and why are we suffering from an increasingly unacceptable level of unreliability ? Unfortunately, the answer is quite simple. Software engineering is a branch of the fashion industry as opposed to the engineering industries. If you are a software engineer and somewhat offended by this statement, consider the fact that software engineering is characterised by an almost complete absence of any systematic form of measurement. This has led to unconstrained creativity, unsupported beliefs and correspondingly erratic if any, progress. As a result we have hundreds of programming languages, hundreds of paradigms and essentially the same old problems. In the '70s, structured design and programming were going to make all the difference; in the '80s, it was the turn of CASE and in the '90s it appears to be object-orientation and formal methods. In each case, the paradigm arises without measurement, subsists without analysis and usually disappears without comment so that we are unable to learn from the experience, the fundamental basis of good engineering. This is essentially the hallmark of an immature engineering discipline. Systematic progress is effectively disabled because marketing shouts whereas measurement only whispers. Somehow, in the next few years, before we do something really stupid like programming a nuclear reactor safety system in an appallingly defined language such as C++, (as I know is already happening - I've seen it with my own eyes), we have to do something about this.

This almost complete absence of measurement has led to a number of beliefs becoming firmly entrenched, many of which are fundamental to the way we think about software and most of which are threatened by recent systems reliability data, some terminally. Let us go through some of these.

Belief: programming language is an important factor in system reliability

The choice of programming language excites hot debate amongst programmers. We all extol the benefits of our favourite programming language whilst denigrating other languages less attractive to us. In truth, published data from around the world of which Table 2 is a subset shows that there is no clear relationship between programming language and the defect density of systems implemented in that language. Ada, for example, supposedly far more secure than other languages produces systems of comparable defect density. In contrast, C is reviled by many safety-related developers and yet it is responsible for some of the most reliable systems ever written. We can conclude that *programming language choice is at best weakly related to reliability.*

Source	Language	Errors / KLOC	Life-cycle
Siemens - operating systems	Assemblers	6-15	Post-del.
IPL - language parser	C	20-100	**PRE**-delivery, therefore higher
NAG - scientific libraries	Fortran	3	Post-del.
Air-traffic control	C	0.7	Post-del.
Lloyds - language parser	C	1.4	Post-del.
IBM cleanroom	Various	3.4	Post-del.
IBM normal	Various	30	Post-del.
Loral - IBM MVS,	Various	0.5	Projected, not actual !
NASA Goddard satellite planning studies	Fortran	6-16	Post-del.
Unisys communications system	Ada	2-9	Post-del.
Ericsson telephone switching	?	1	Post-del.
Average for applications	all	25	Post-del.
Average for US and European applications	all	5-10	Post-del.
Average for Japan	all	4	Post-del.
Motorola	?	1-6	Post-del.
Operating systems, Akiyama, 1975	Assembly	20	Post-del.

Table 2: Defect densities from various studies in a number of languages.

Belief: Object-orientation makes a big difference to reliability

All around the world, massive investments are being made in object-orientated technology in the belief that it is so self-evidently better that data is unnecessary. In fact, recent data casts serious doubt on its many promises. In a recent study comparing two similar systems of similar size, (around 50,000 lines each), one in C and one in object-designed C++, the resulting defect densities were shown to be around the same at 2.4 and 2.9 per 1000 lines respectively, [2]. Far more serious however was the fact that this study showed that the overall corrective maintenance cost, (the cost of fixing bugs after the software has been released and already 40% of the entire software life-cycle cost), had increased by a factor of nearly three. Other studies have similar reservations, [3]. This study may simply reflect deficiencies in C++, but there appears to be little if any similar data for other object-oriented languages such as Smalltalk or Eiffel. To conclude, such object-oriented data as there is seems to extend the earlier conclusion that language has little effect on reliability, object-oriented or not and that the massive drive to object-orientation is another giant leap sideways in which the software industry appears to specialise.

Belief: Formal methods will be the big step forward

A great deal of effort has been expended in recent times in support of the conjecture that formal methods, (the use of mathematically based notation in both specification and/or implementation), will lead to significant and indeed dramatic improvements in software reliability.

Formal methods unquestionably have a place, but where and to what extent ? In a recent study of an air-traffic control software system comprising both formally and non formally developed components, [4] found that pre-delivery faults were not significantly affected by the use of formal

specifications, whereas post-delivery faults were significantly and beneficially affected by a factor of around 3. In other words, had testing not been effective in this development, the resulting system would not have had particularly high reliability in spite of the use of formal specifications. We can deduce that formal specification appeared to work only *in consort* with strict objective test targets to produce a delivered system of very high reliability and a defect density of around 0.7 per KLOC. Serious defects occurred some 25 times less often.

To summarise, formal methods on their own should not be expected to improve matters much in general, in that with current software engineering capabilities, it does not appear to be addressing the weakest link. Only if used with other techniques such as effective testing will their benefits be realised and even then, the scale of improvement appears to be modest - an important step forward but not a giant leap.

Belief: ISO 9001 and level of integrity have a big effect on intrinsic software product quality

Much has been written of process initiatives such as ISO 9001, the belief being that the existence of a consistent software process will lead to software products of higher quality. As always, so much money has been invested that the belief becomes an end in itself. In fact, if we define intrinsic software quality by the relative occurrence of statically detectable fault, standards transgressions and absence of overly complex components, this belief is manifestly wrong as discussed in detail by [5]. In fact, in studies of statically detectable fault in several languages, he showed that there is little relationship between the presence of such fault and either the level of integrity of the corresponding code or its process certification. Standards transgressions also show no particular pattern.

Belief: In any system, the small well-structured components are the most reliable.

Of all the beliefs challenged in this article, the belief that decomposing a system into small components improves reliability, is savaged to the point of no return by numerous data sets acquired over the last few years in various languages, and yet it has been unchallenged for 25 years and indeed has underpinned much of modern software engineering, [6]. The languages include Ada, C, C++, Fortran 77, Pascal and various dialects of assembler and cover application areas as disparate as communication systems, operating systems, numerical analysis libraries, databases, aerospace applications and so on. The data is unanimous and no conflicting data has yet appeared. In other words, the evidence is overwhelming.

The apparent language independent behaviour of this phenomenon led [6] to suggest that the effect is to do with the way humans manipulate symbolic data rather than any particular representation, and to derive a mathematical model of defect density based on memory breakdown in the human short and long term memory. This model predicts the observed defect density v. component size behaviour well over a very broad range of component sizes as evidenced by Figure 2.

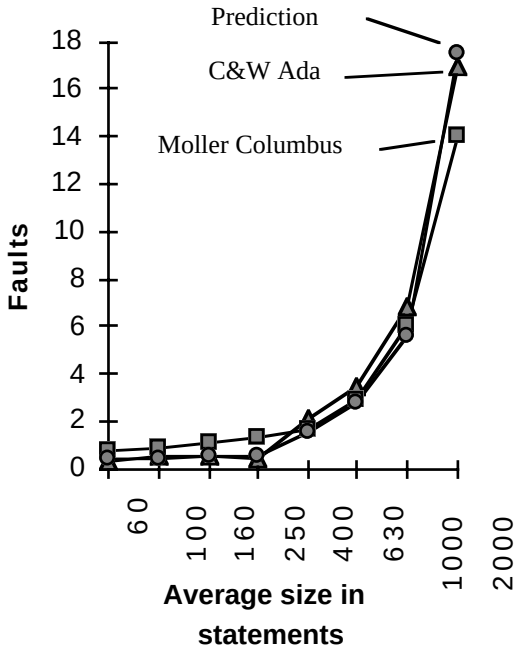

Figure 2: Normalised defects versus average component size in statements for observed data from Ada and assembler systems compared with a prediction using the human memory model of [2].

Of particular interest in the observed data is that normalised defects appear to be only logarithmic with average component size. Of course they would have to be linear or worse for large components to be proportionately worse than small components, so the belief that systems made from small components are intrinsically more reliable should be treated as simply wrong, although it seems a centre-piece of most object-oriented implementations for example. The problem becomes glaringly obvious when the data is displayed as a defect density curve versus average component size as shown in Figure 3.

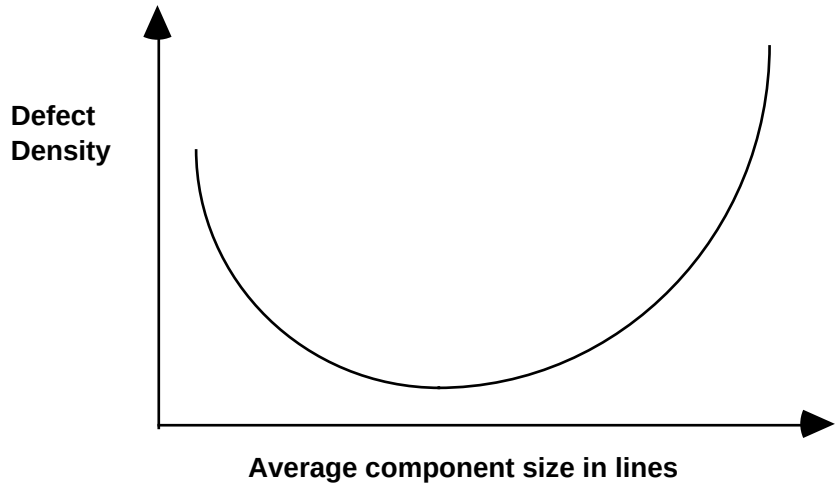

Figure 3: The U-shaped nature of the defect density curve compared with average component size. The position of the observed minimum appears to be insensitive to language and occurs in the range 150-250 lines. In other words, in any system, the 'medium' sized components are by far the most reliable. This curve has been substantiated for Ada, C, C++, Fortran, Pascal and various assemblers over a very disparate range of application areas and appears to be a fundamental property of software systems.

Belief: Re-use will improve reliability

The logarithmic behaviour reported in the previous section has some interesting implications. For example, if an existing system is reduced in size using re-use, a simple calculation suggests that the overall system reliability will initially get *worse* before it begins to get better at very high levels of re-use (typically that which reduces system size by > 70%). Furthermore, if an existing component is re-used in a new design, the system reliability may also be prejudiced unless the re-usable component is of at least as high quality as the new system, it has been noted on numerous occasions that modified components, as re-usable components usually are, may cause more problems than new components. The implication of these data and observations is that predicting the effects of re-use is in fact a very complex question to which there appears to be no simple answer. In real life, re-used components which were safe in their original environments were responsible for the grave failures in Ariane 5, (see below), and also the notorious Therac-25 incident in which a computer-controlled radiation therapy machine massively overdosed six people.

What can we do about it ?

I stated rather tantalisingly at the beginning that a significant number of major failures could have been avoided. Various studies suggest that around 40% of all software failures could have been detected before the code was even compiled using techniques we already know to do. As an example, Figure 4 shows the distribution of about a 100 different statically detectable faults in commercially released C applications in a study done between 1992 and the present day using QAC, a deep-flow static analysis tool from Programming Research Ltd. Very similar distributions occur in other languages and result from the fact that all programming languages have well-known loopholes into which programmers fall again and again. These can be broadly categorised into:

a) Items which the programming language committee was unable to define unambiguously for political or technical reasons, (explicitly undefined behaviour), for example reliance on variables which have not been initialised.

b) Items the programming language committee forgot to define, (implicitly undefined behaviour). These are language dependent.

c) Inaccuracies in the programming language standard document itself.

d) Items which are perfectly well-defined but which programmers consistently mis-use. An example of this category is the ability to leave a decision structure incomplete.

The sad fact is that many of these loopholes could have been avoided when the source code first appeared. *Every* defect appearing in Figure 4 could have been avoided as they were detected simply by inspecting the source code and not running it.

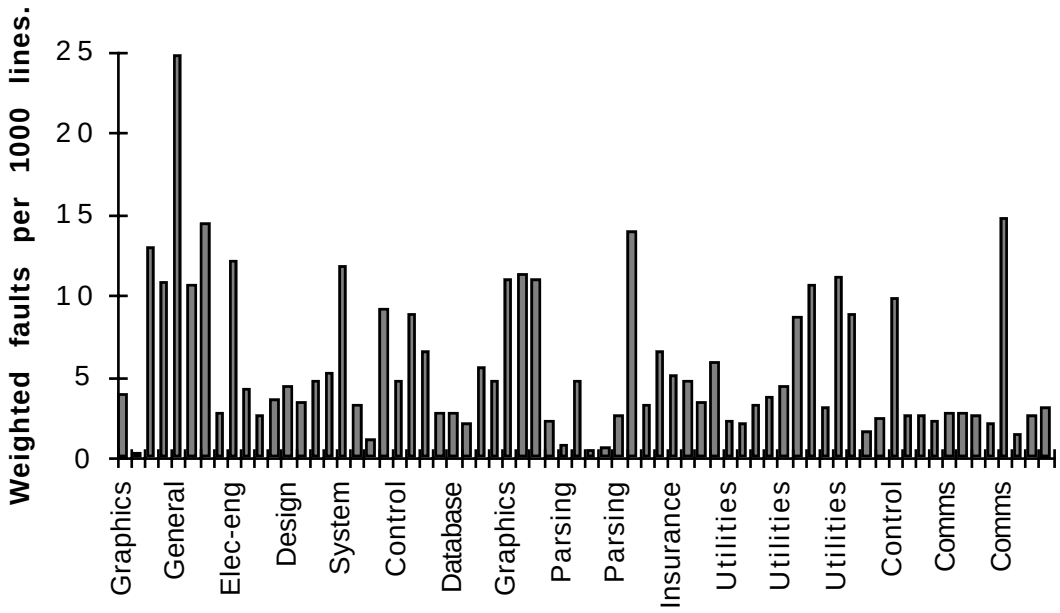

Figure 4: The distribution of statically detectable faults per KLOC (1000 lines of code) in commercially released applications around the world measured between 1992 and the present day. The codes were generated in the last twenty years and come from many kinds of applications and industries around the world. Several million lines are represented.

Examples of avoidable failures:

A prime example of this is the recent spectacular failure of Ariane 5. In this case, the problem was very simple. The programmers took a 64-bit floating point number and jammed it into a 16-bit integer without checking to see if it could be correctly represented or not. It couldn't, although the same component had been used successfully in Ariane 4 where it turned out that it could because Ariane 4 had a different flight trajectory. The language used was Ada, the darling of the safety-critical industries which just goes to show that if programmers inadvertently screw things up, the language isn't going to stop them. The reader may correctly divine that I am a little cynical about this. The reason is that this particular mistake has occurred on many occasions over the years and we still persist in doing it. As a further example, a form of it was responsible for the Shuttle Challenger failing to rendezvous with the damaged Hughes F/6 Intelsat satellite in 1992 at a reported cost of around $400 million. In this case, the programmers concerned had inadvertently fitted a 64-bit floating point number into a 32-bit floating point number. The programming language again did not prevent it. The resultant halving of precision amounted to too large a distance in space to complete the rendezvous successfully.

So why do we allow such well-known problems to occur again and again when we know how to prevent them ? Beats me.

References

1. Wayt Gibbs, W., *Software's Chronic Crisis,* in *Scientific American* 1994, p. p. 72-81.

2. Hatton, L., *Software Failure: the huge cost, the avoidable and the unavoidable*. to appear in 1997

3. Cartwright, M. and M. Shepperd. *Maintenance: the future of object-orientation*. in *CSM'95*. 1995. Durham, England:

4. Pfleeger Lawrence, S. and L. Hatton, *How do formal methods affect code quality ?* To appear in IEEE Computer, 1996.

5. Hatton, L., *Safer C: Developing for High-Integrity and Safety-Critical Systems*., 1995, McGraw-Hill, ISBN 0-07-707640-0.

6. Hatton, L., *Why is the defect density curve U-shaped with component size ?* IEEE Software, 1997. (March).

Dr, Les Hatton, C.Eng. is a software consultant in high-integrity and safety-critical systems and is a director of Programming Research Ltd. This article is abstracted from a new book entitled "Software failure: the huge cost, the avoidable and the unavoidable", to appear shortly. He can be reached at: Oakwood Computing, Surrey, KT3 3AJ, U.K., Tel/Fax: 0181-336-1151; lesh@oakcomp.demon.co.uk.