Title: **Some notes on software failure**

Date: **24/Oct/2001**

Author: **Les Hatton**

The problem

Software failure manifests itself in at least three important ways:-

- Process failure. Here the process of producing software fails in some fundamental way so that the wrong system is produced, or the right system is produced but very late, or even no system is produced at all. This has been a very common source of failure in all types of development. An example is the Taurus Stock Exchange system in the UK as well as a significant number of Government sponsored initiatives. The NHS has been rather blighted here. We are not alone. In the US, a study of $140 million worth of Flight Controls Software Projects 1985-1990 by the Audit Office of the US D.o.D. revealed that 90% was either never delivered or never worked.
- Product failure (cessation). Here the product fails when it is running leading to some adverse behaviour. There are numerous examples of this with a significant number of billion dollar failures occurring around the world in the 1990s, the first probably being the AT&T failure of January 1990 when a single mistake took down the entire US long-distance telephone network for 9 hours. In the last two years, most major car companies have had very expensive recalls because of mistakes in software controlled systems leading to unacceptable failures.
- Product failure (misleading results). Here the results of scientific research are erroneous because they are computer simulated by defective software. As an example of this, the main mathematical technique used for oil & gas exploration is fundamentally damaged by software failure, (Hatton & Roberts (1994), Hatton (1997)), with an unexpected drop to one significant figure of accuracy instead of the expected four and the necessary three. The evidence suggests that many other numerical simulations in science may be similarly affected.

In aggregate, the cost of failure is now such as to be unquestionably damaging to the UK economy (as well as all other major economies) and a resolve to improve matters would be central to the global performance of the UK economy in IT and any dependent activity. It is a genuinely strategic issue.

Aggravating factors

- Size. Software systems are growing by a factor of two every 18 months in consumer embedded systems. Today we have around 3,000,000 source lines in a car.

•Coupling. Networking such systems leads to new and poorly understood failure modes through component interaction.

•Chaotic behaviour. Software systems tend to fail chaotically. A small change in a program can lead to a small or large effect in the run-time behaviour with little to guide us on which.

•Cost of failure. The 1990s saw the first of the billion dollar failures and a number of repetitions.

•Reduced time to market. The pressure of modern development usually leads to software being released too early.

Some ancillary comments

•Perfection is not an option. If a software product has less than one fault which fails per 1000 lines of source code in its entire life-cycle, it is about as good as has ever been systematically achieved. In addition, 5-10% of these failures will be deemed significant. This means that in a million line development (fairly typical today), even with a state of the art process, the product will exhibit around 1000 defects in its life-cycle of which 50-100 will be serious. A reasonable product could be expected to be about 5 times worse.

•Extensive testing does not imbue a product with reliability. The discovery of many defects found during testing is highly correlated with the fact that many more will appear after release. Reliability must be designed in; it cannot be tested in.

•A high percentage of faults take a very long time to appear. In a famous experiment reported at IBM in 1984, (Adams (1984)), a third of all faults took longer than 5000 execution years to fail. In addition, it was found that every 7 faults corrected led to the injection of one new one at least as severe due to unintended side-effect. This experiment proves conclusively the futility of dynamic testing as a method of improving reliability as intimated in the previous point.

•The use of formal mathematically based notations help but seem to give only a modest improvement (a factor of about three in defect density) which although it attacks faults injected during design, the improvements can easily be swamped by other classes of injected fault such as implementation faults, (Pfleeger and Hatton (1997)). As a result, you cannot rely on such techniques alone.

•Software is cursed with unconstrained creativity. Many new paradigms are introduced which have no effect on reliability principally because they exist in a measurement vacuum and are essentially fashion-based. For example, the very popular and ubiquitous OO (Object-Oriented) methodology appears to have some fundamental flaws leading to longer fault correction times not

shorter, (Hatton (1998)).

•The classic engineering paradigm of control process feedback, ("do not make the same mistake twice"), is almost completely absent from software engineering with repetitive failure modes common, (i.e. the same fault failing repeatedly because it cannot be located from the evidence available). This is a direction consequence of the almost complete absence of measurement and analysis coupled with the poor understanding of prediction and diagnosis in software engineering systems, (Hatton, (2001)).

•Software process research leading to initiatives such as the CMM is incomplete. One of the most reliable software components in history is the Linux kernel. Linux is developed in the complete absence of numerous processes stated as significant in the CMM, (Capability Maturity Model of the Software Engineering Institute at Carnegie-Mellon, now a US D.o.D. Standard). The CMM in spite of its immense popularity particularly in the US, appears neither necessary nor sufficient.

•A significant percentage of software failures, perhaps as high as 40% could have been avoided using techniques we already know how to do. For shame we can do better than this.

•In software engineering, technology problems do not appear to be the most significant, this role appears to be played by educational problems. There are a significant number of experiments around the world which show that variations in individual engineers generally dwarf variations in technologies, (Prechelt, Tichy, (1995-)).

Summary

Software engineering is unquestionably a huge benefit to society. There remain very significant problems however and the current cost of failure is simply enormous. In an age where jobs are threatened by outsourcing software development to locations like the software factories of India, higher reliability software and the consequent dramatically reduced costs can make or break an economy. The difference in price between developing in India and developing in Britain is less than the cost difference between software reliability as it stands today and how good we could make it with a relatively modest but concerted effort.

References

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

Hatton, L. and Roberts A. (1994) "How accurate is scientific software ?", IEEE Transactions on Software Engineering, 20(10), p. 785-797.

Hatton, L. (1995) "Safer C: developing software in high-integrity and safety-critical systems", McGraw-Hill, ISBN 0-07-707640-0; Chapter 1.

Hatton, L. (1997) "The T experiments: errors in scientific software", IEEE Computational Science and Engineering, 4(2), p. 27-38.

Hatton, L. (1998) "Does OO sync with the way we think ?", IEEE Software, 15(3), p. 46-54.

Hatton, L. (2001) "Exploring the role of diagnosis in software failure", IEEE Software, July.

Pfleeger, S.L. and Hatton L. (1997) "Investigating the influence of formal methods", IEEE Computer, 30(2), p 33-43.

Prechelt, L., Tichy W. (1995-) http://wwwipd.ira.uka.de/~prechelt and ~tichy.

