"Characterising the diagnosis of software failure "

by

Les Hatton

Oakwood Computing, U.K, and the Computing Laboratory, University of Kent. U.K.

Version #1.3, 12th. July, 1999

Abstract

Software systems are characterised uniquely amongst engineering systems by repetitive failure. This paper discusses some of the reasons for this unhealthy situation and highlights poor diagnosability as a pivotal factor. This is to a large part an educational problem,. Software engineers are encouraged to believe that their ambition to remove defects is matched by their capability. As a result, the possibility of software failure is often not in contemplation during software design and as a result, when the system inevitably fails, the responsible fault or faults cannot be diagnosed. This situation is getting worse as systems become larger and more tightly-coupled. Drawing on many examples, this paper highlights the need for a change of attitude to software failure and the overwhelming need for improved diagnostics.

Repetitive failure in software systems

Software systems are unique in engineering systems generally in that they continue to exhibit and are frequently dominated by repetitive failure. More than 15 years ago, Ed Adams published his by now famous study at IBM, (Adams 1984), showing that a very small percentage of faults, (only around 2%), were responsible for most of the observed failures. In other words his data showed that his systems were dominated by repetitive failure.

What do we mean by fault and failure ? In essence, a fault is a static property of a design or a piece of code. In other words, an engineer can simply inspect the design or code and declare that it is inconsistent with his or her understanding of the intended behaviour of the software. The engineer is not running the code, he or she is simply extrapolating from their knowledge of the programming language and design methodology and their knowledge of the intended behaviour of the system to infer that the design or code is incorrect.

In contrast, a failure is an observed property of the run-time behaviour of the system. It occurs when the actual behaviour of the

system deviates from the expected behaviour. The relation between the two in simple terms is shown in Figure 1

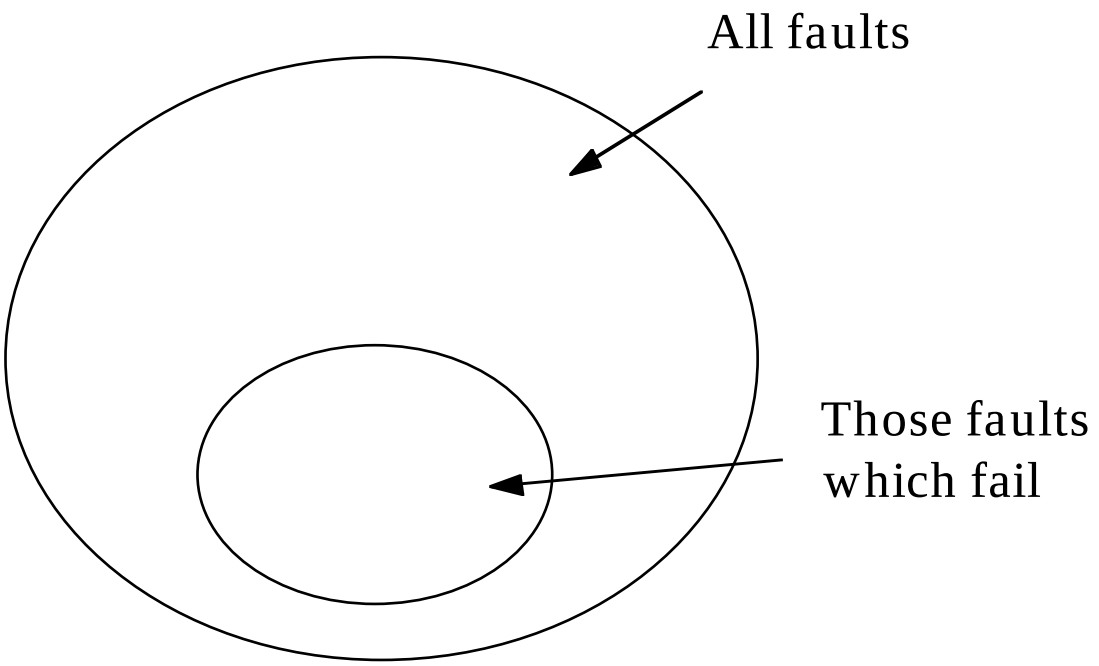

Figure 1. The relationship between fault and failure.

In essence, every failure is caused by at least one fault, but not all faults fail during the life-cycle of the software. In fact, Ed Adams showed that fully a third of all faults failed so rarely that to all intents and purposes they never failed at all in practice. It is quite easy to envisage a fault which does not fail. Suppose the fault exists in a piece of code which is actually unreachable, then the fault could never fail. (Note that it is not unknown to have unintentionally unreachable code in large systems.) In general, however, under some set of conditions, a fault or group of faults will combine to cause the system to fail.

As will be discussed at length later in this paper, the essence of diagnosis is to trace back from a failure to the culpable fault or faults. It is important to note that we have no systematic model which allows us to go in the other direction i.e. to predict failure from a particular fault or group of faults. This is the province of the discipline known as “software metrics”. Here we attempt to infer run-time behaviour and in particular failure occurrence from some

statically measurable property of the code or design. For example we might like to say that components with a large number of decisions are unusually prone to failure. We have in general so far failed to produce any really useful relationship and the area is very complex, (Fenton 1991) , (Hatton 1997). This asymmetry is particularly clear when considering the difference between code inspections and traditional testing. Code inspections address the whole fault space of Figure 1. Consequently a large percentage of faults revealed during code inspections would never actually cause the system to fail in a reasonable life-cycle. In spite of this, the dramatic effectiveness of code inspections is unquestionable, (Gilb and Graham 1993). In contrast, when traditional testing finds a problem at run-time, this by definition is a failure and lies in the small subset of Figure 1. (In fact, we can predict that code inspections would be even more effective on systems which build up many execution years, such as modern consumer embedded systems, because the subset of faults which fail is much bigger in this case.)

There have been many attempts at understanding failure from static properties of code such as complexity measurements, often mistakenly called metrics, but the primary reason why we have no systematic model which can predict a failure from a fault is that *the process is fundamentally chaotic*. In conventional engineering, we work in the linear zone where stress is linearly related to strain and the behaviour of the engineering system at 'run-time' is much more predictable. Software is in general fully chaotic. Let's consider two classes of example. The first illustrates incredible sensitivity and the second illustrates equally incredible insensitivity.

The AT&T 1990 outage and the DSC outage

These have been discussed in various places, e.g. (Watson 1992), but in each case, a tiny coding change led to a catastrophic failure. In the AT&T outage of 1990, a single misplaced line of code in 3 million lines of network management code downed the entire long-distance network of the U.S. for several hours. The DSC outage was caused by just a three line change.

Insensitivity in a numerical weather prediction model

This example represents the other end of the sensitivity spectrum and is discussed in more detail in (Hatton 1995). In this case, in my first professional job in 1974 at the U.K. Meteorological Office, I was re-writing a ten-level numerical weather prediction system in a new coordinate system. The computational parts of this program were written in IBM OS/360 assembler for efficiency. Half way through this several month project, I found a terrible fault. At every other time-step, due to an accidental transposition of two assembler statements, all of the non-linear terms of the equations were being unintentionally zeroed. To cut a long story short, the non-linear terms in the governing Navier-Stokes equation are responsible for *all* of the weather. Without them, the atmosphere simply degenerates into hydrostatic equilibrium. Restoring the terms caused hardly any observable difference even over the longest predictions. In this case, the end-result was almost completely insensitive to what I would have thought was a fundamental change. (An ensuing widely-published memo meant as an innocent joke and suggesting that all the computational terms be removed for extra speed, led to the abrupt end of my career as a civil servant !)

So we have seen that the unpredictable sensitivity of failure to fault leads to chaotic behaviour. Let us return to the inverse process of diagnosis. We can ask the question why would software systems be dominated by repetitive failure ? After all, other engineering systems are not. If bridges failed the same way repeatedly over some period of time, there would be a public outcry. In fact, if you go far enough back in time, bridges did fail repeatedly, (and there was a public outcry). For example, in 1879, the Tay bridge in Scotland fell down in a strong gale. It was poorly designed and built and the wind blew it over causing the deaths of 75 people in the train it was carrying at the time. More or less as a direct result of the ensuing public outcry, the Forth bridge in Scotland built a little later is almost embarrassingly over-engineered. If the earth is ever hit by a comet, we will have to hope that it hits the Forth Bridge first. It will very likely bounce off. This process of engineering maturity whereby past mistakes were gradually avoided in future designs, has led to the almost complete disappearance of repetitive failure, not

only from bridges, but also most other areas of engineering. Some two thousand years ago, the Carthaginian general Hannibal crossed the European Alps using elephants to carry his stores. This involved building a large number of bridges. Hannibal as a matter of good engineering practice always made the bridge engineers cross first on the heaviest elephants, thus concentrating their minds wonderfully.

The single exception is software. It is easy to think that this is because software development is only around 50 years old, however, this is overly generous and ignores the lessons of history. The fact is that software engineering is gripped by unconstrained creativity and everybody is having far too good a time to be bothered by such things as the elimination of repetitive failure. It is hoped that this attitude will not persist for too long

There are all sorts of reasons why software fails repeatedly. Probably the most significant one is that we do not in general learn from our mistakes. The standard method of engineering improvement has been known for a very long time and is shown in Figure 2. It used to be called common sense but in these enlightened times, with the addition of a little mathematics, it is known as control process feedback.

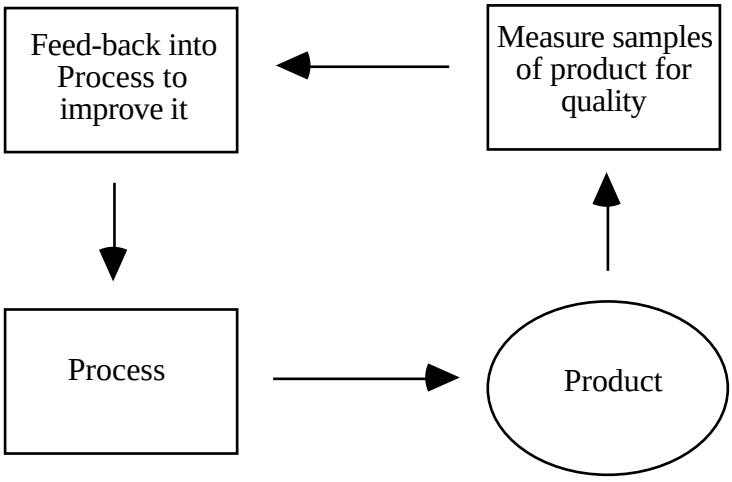

Figure 2 The 'silver bullet' of engineering. It embodies the extraordinarily simple principle that it is not a sin to make a mistake, it is a sin to repeat one. Careful analysis of failure reveals the all important clues as to how to avoid it in future.

Regrettably, use of this simple principle is not widespread in software engineering. It is almost as if it is too obvious, a drawback it shared with Darwinian evolution for much of the last 200 years or so. The writer and technological seer Arthur C. Clarke once said that "any sufficiently complex technology is indistinguishable from magic". I would like to paraphrase this to say that *any sufficiently simple technology is also indistinguishable from magic*. Darwinian evolution and control process feedback are two outstanding examples of this principle in action. They are apparently so obvious that they shouldn't be right. However, probably the most important characteristic these two share, is that improvement is inexorable but it also takes time. This is particularly relevant in these days of "reduced time to market", (also known as "don't test it as much"),

Another widespread source of repetitive failure in software systems is caused by imprecisely defined programming languages and the absence of any means of avoiding these problems in most organisations. This is exacerbated by the process of *language standardisation*. We would all agree that standardisation is an important step forward in engineering maturity, however, if the process of standardisation ignores historical lessons, then this may well be worse than useless. Language standardisation suffers from two important drawbacks as practised today. First of all, language committees (and I've sat on a few in my time), have an irresistible temptation to fiddle. They will persist in adding features which seem like a good idea at the time, without any notion as to whether they will work or not. Of course, this is normal in engineering. It is similar to the role of mutation in Darwinian evolution. What is not normal however is the second drawback. This embodies the opposing principle to control process feedback. It is called "backwards compatibility" and is often expressed in the hallowed rule that "thou shalt not break old code". So drawback one guarantees the continual injection of features which may or may not work, (most don't) and drawback two guarantees that you can't take them out again. In other words it is a technique whereby learning from previous mistakes is guaranteed *not* to take place. In backwards compatibility, you take as a starting point all the failure modes which have occurred so far and then add new and poorly understood failure

modes. We call the result a modern programming language. If other engineering disciplines pursued this doctrine, hammers for example would have micro-processor controlled ejection mechanisms to cause the head to fly off randomly every few minutes as they used to about 40 years ago when made with wooden handles. Not surprisingly, they were redesigned fairly quickly.

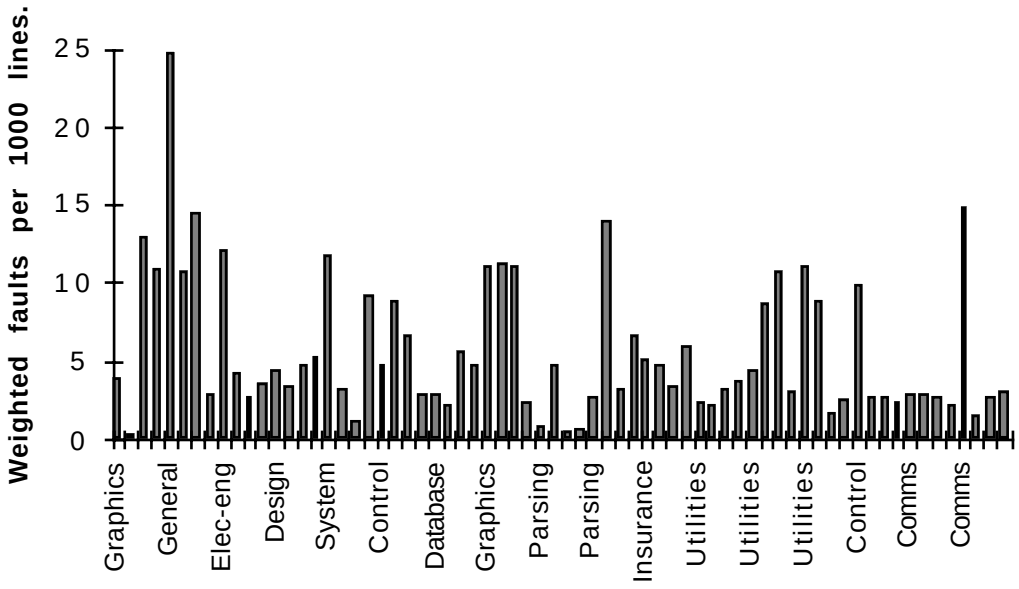

Figure3: Weighted fault rates per 1000 lines of code for a wide variety of commercially released C applications plotted as a function of industry.

The effects of this can clearly be seen in Figure 3, an analysis of dependencies on a significant class of software faults occurring in a large study of commercial C systems, (Hatton 1997). This class of faults is widely published, has been known for the best part of 10 years, in some cases 20 years, is known to fail and yet nothing much has been done to prevent it. I would like to say that the latest incarnation of C due for standardisation in the next year will improve matters, but it does not for the reasons given above. In a number of important ways, it is fundamentally less reliable. We should not be too hard on C because precisely the same thing occurs for other commonly used languages. The sad thing is that no standardised programming language can ever improve - each one is doomed to be progressively devoured by poorly-understood and often unnecessary complexity. It is no wonder then that independently written programs tend to suffer from dependent

failure, (Knight and Leveson 1986), when software in general is riddled with a growing number of known repetitive failure modes which we take almost no action to avoid. I look forward to a time when we can repeat the Knight-Leveson N-version experiment in languages from which repetitive failure modes have been systematically eliminated using the mechanism in Figure 2. I very much doubt if I will see it during my career.

A third reason why software is full of repetitive faults is the main subject of this paper. This is the problem of diagnosis. Diagnosis is the mechanism of deducing the fault or set of faults which were the cause of an observed failure. If a system fails and diagnosis fails to reveal the fault or faults which caused it, or perhaps only yields a subset of faults which caused it, (bearing in mind one of the findings of (Adams 1984), whereby 1 in 7 defect corrections is itself faulty), the failure will occur again in future in some form. In other words, the inability to diagnose a failure inevitably results in that failure becoming repetitive.

In the following, we will attempt to analyse why diagnosis often fails.

Quantifying diagnosis

Two essential parameters determine how easy a software failure can be diagnosed. They are *diagnostic distance* and *diagnostic quality*.

Diagnostic distance

In essence, diagnostic distance is the ‘distance’ in the system state between a fault being executed and the resulting failure manifesting itself. No natural candidate conveniently presents itself but it can be reasonably visualised as the number of changes of state which take place between the execution of the fault and the observation of failure. The greater the number of changes, the more difficult in general it is to trace the failure back to the fault, i.e. to diagnose it. Let us consider some simple examples from different languages and different systems..

Consider Figure 4. This shows a typical “stack trace” resulting from a core dump generated in this case by a C program. In short, the program has tried to look at the contents of address zero, a fatal mistake in C leading to an immediate program failure In other words, the ‘distance’ between the fault firing and the failure occurring is very short. This coupled with the detailed nature of the stack trace points unerringly back at the precise location of the fault. Not surprisingly, these failures are very easy to diagnose and as a result they do not commonly occur in released systems. (c.f. the Appendix of (Beizer 1990)).

```
Dereference pointer contents 0x0 at
strlen(…) called from
line 126 of myc_constexpr.c called from
line 247 of myc_evalexpr.c called from
line 2459 of myc_expr.c
```

Figure 4 A typical stack trace automatically produced by a reasonable C compiler, (in this case the estimable GNU compiler), at the point of dereferencing a pointer containing the address zero.

We can contrast this failure with the next one as shown in Figure 5. This one occurred in a Fortran 77 program some years ago, (Hatton, Wright et al. 1988), but exactly the same problem manifests itself in other languages. In this case, a comparison of real variables behaved slightly differently on different machines, a common problem. The effect was a drop in the significance of agreement from 4-5 decimal places to 2 decimal places in some parts of a dataset which formed part of an acceptance test. The requirement was for at least 4 significant figures of agreement between two different specified target computers. This line of code was buried in the middle of a 70,000+ line signal processing package containing amongst other things, Fast Fourier Transforms and various other signal processing algorithms. From the point of failure, it took the author and a colleague on and off some 3 months to trace back to this line of code.

```
…
if ( tolerance .eq. acceptable_tolerance) then
…
```

Figure 5 A comparison of real valued variables. This is broken in just about every programming language in existence.

In this case, the diagnostic distance between the fault and the failure was sufficiently large to lead to an exceptionally difficult debugging problem.

Other examples of large diagnostic distance can be seen in dynamic memory failures in C, whereby dynamic memory is allocated, corrupted at some stage and rather later on leads to a failure, and also in several aspects of OO implementations. Another way of describing this is to categorise it as *non-local behaviour*.

Diagnostic quality

In contrast, diagnostic quality refers to how well the diagnostic distance is signposted between the state at which the fault executed and the state at which the software was observed to fail. To give a simple example of how influential this can be, remove a significant token from the middle of a computer program, for example, a ‘,’ from a C program. When such a program is recompiled, the compiler will diagnose the absence of this token usually with a reasonably comprehensible error message. An example from the estimable GNU compiler is shown in Figure 6.

```
…
myc_decl.c:297 parse error before ‘name’
…
(lots of increasing more exotic messages)
…
myc_decl.c: 313: warning: data definition has no type or storage class
```

Figure 6 Part of the fault cascade on one of the author’s programs which has had a significant token, in this case a ‘,’ deliberately removed. Warnings near the point of

occurrence are relatively easy to understand. Later warnings in the cascade rapidly get more and more irrelevant.

Note that the missing token leads to a cascade of error messages starting at the location of the fault, but which rapidly degenerate into entirely spurious messages, until the compiler eventually gives up in disgust. This is a classic problem in compiler design for all programming languages. Now imagine trying to diagnose the missing token *from the last compiler warning*. Clearly, this can be very difficult. The *full* error cascade signposts the fault-failure path. Removing part of the signposting can fatally cripple the ability to diagnose the fault from the failure.

Such cascades are becoming common in real systems as they become more tightly coupled and larger. Consider the example in Figure 7, quoted by (Mellor 1994), in an excellent discussion of this topic.

- MAN PITCH TRIM ONLY, followed in quick succession by:-
- fault in right main landing gear
- at 1500ft., fault in ELAC2, (one of 7 computers in the EFCS)
- LAF alternate ground spoilers 1-2-3-5 (fault in Load Alleviation Function)
- fault in left pitch control green hydraulic circuit
- loss of attitude protection (which prevents dangerous maneouvres)
- fault in Air Data System 2
- autopilot 2 shown as "engaged" despite the fact that it was disengaged, and then finally,
- "LAVATORY SMOKE", indicating a (non-existent) fire in the toilets.

Figure 7 The diagnostic sequence appearing on the Primary Flight Display of an Airbus A320, Flight AF 914 on 25th August 1988 The faults were not spurious - the aircraft had difficulty getting its landing gear down and had to do three passes at low altitude by the control tower so that the controllers could check visually.

Imagine trying to diagnose the landing gear problem from the last error message in this case ! In fact, this led to a repetitive failure mode. At least one more incident occurred, (on the 19th November, 1988) and a further 9 months went by before a fix was made.

Training engineers to provide adequate signposting so that faults can be diagnosed from failures is a problem of education. We all too often train software engineers that removing the last defect from a piece of software is a sensible option. This encourages over-

optimism and leads to inadequate preparation for the inevitable failure. The diagnostic link between failure and fault is simply not present and the fault becomes difficult if not impossible to find. Even when it is present during testing, it is often removed for reasons of space from the released system, crippling any ability to diagnose failure properly when it occurs. Consider the following examples of risible warnings issued in commercial systems.

"*Please wait ...*", (a FMGS failure on an Airbus A340 in September 1994, (AAIB 1995). To the author's knowledge the cause has still not been found.).

"*System stressed ...*", (a cash register system in a public bar. The printer had run out of paper it later transpired, (Hatton 1999)).

"*More than 64 TCP or UDP streams open ...*". (a G3 Macintosh running OS8.1. It turned out that the modem was not switched on, (Hatton 1999)).

Of course the ultimate in poor diagnostics is no warning at all. At the time of writing, there is considerable public debate in the U.K. about the Chinook helicopter crash in 1994 on the Mull of Kintyre in Scotland, (Collins 1999). The crash killed 30 people including both pilots and a number of very senior security people from Northern Ireland. The cause of the crash was blamed on pilot error. Prior to this incident, there had been a number of concerns raised about the quality of the FADEC software controlling the engines of this aircraft, including a design flaw which had precipitated the near destruction of another Chinook in 1989. The requirements for a verdict of pilot error must be 'no doubt whatsoever' and yet the essence of the U.K. government's argument seems to be that because the crash investigators found no evidence that the FADEC software failed, then it must have been the pilots. However, as any PC user knows, most PC crashes leave no trace, so that on reboot, there is no evidence that anything was ever wrong. It is therefore grievously wrong to equate no diagnosis with no failure.

Unification

We can put together these two different parameters to summarise the diagnosis problem as shown in Figure DIAGNOSE.1.

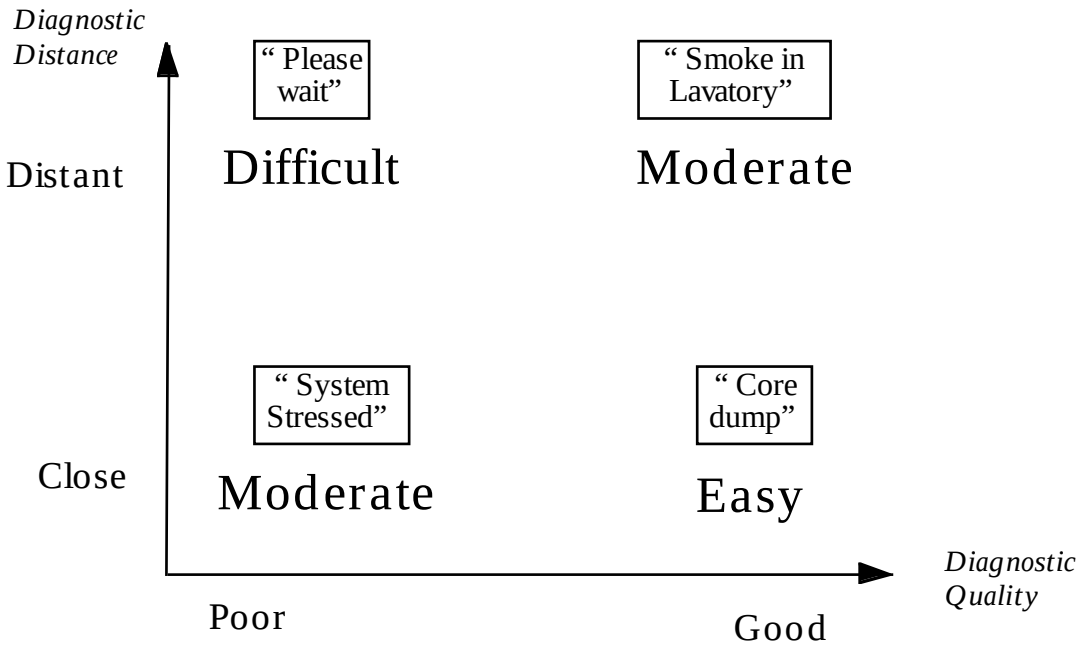

Figure DIAGNOSE.1 Diagnostic distance v. diagnostic quality with some of the examples from the text marked to signpost the space.

Here we can see that when diagnostic distance is considerable, then unless diagnostic quality is good, we are presented with a very difficult diagnostic problem and one that in general will fail. The 'Difficult' part of this figure can be seen as a source of repetitive failure. With such a problem we can ameliorate it either by improving the diagnostic quality or by reducing the distance between the point where the fault executed and the point where the failure was observed. In practice the latter is much easier than the former.

Conclusions

Software engineering in the 1990s is producing more and more systems where diagnostic distance is large. Networking is a classic method for increasing diagnostic distance for example. Embedded systems are similarly difficult to diagnose. Unless we can make rapid progress in improving diagnostic quality, an essentially

educational issue, things are going to spiral out of control and we can look forward to repetitive failure becoming a permanent fixture in software engineering systems. This is clearly an unacceptable position.

Acknowledgements

All sorts of people contributed to this paper. I would particularly like to thank Peter Mellor for his input.

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.

Beizer, B. (1990). Software Testing Techniques, Van Nostrand Reinhold.

Collins, A. (1999). "MPs misled over Chinook". Computer Weekly.

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

Gilb, T. and D. Graham (1993). Software Inspections. Wokingham, England, Addison-Wesley.

Hatton, L. (1995). Safer C: Developing for High-Integrity and Safety-Critical Systems., McGraw-Hill.

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.

Hatton, L. (1999). "Repetitive failure, feedback and the lost art of diagnosis." Journal of Systems and Software.

Hatton, L., A. Wright, et al. (1988). “The Seismic Kernel System - A Large-Scale Exercise in Fortran 77 Portability.” Software Practice and Experience **18**(4): 301-329.

Knight, J. C. and N. G. Leveson (1986). “An experimental evaluation of the assumption of independence in multi-version programming.” IEEE Transactions on Software Engineering **12**(1): 96-109.

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

Watson, G. F. (1992). “Faults & failures.” IEEE Spectrum(May 1992).