"Why is the defect density curve U-shaped with component size ?"

Les Hatton[1]

ABSTRACT

This paper explores the empirical results of a number of recent (and not-so-recent) papers showing that *larger software components are proportionately much* ***more*** *reliable than smaller software components within the same system up to a certain size after which they rapidly deteriorate*. This is strongly counter-intuitive to basic notions of software engineering such as modularisation.

The paper first demonstrates that a *logarithmic* distribution of fault with component complexity closely fits the observed data over a range of component sizes and languages up to around 200 lines or so, (deemed medium size here), after which approximately *quadratic* behaviour is observed. The paper will review mitigating influences for this non-intuitive behaviour before concluding that none is really satisfactory. It then unites this complex behaviour in a simple mathematical model of the physiology of the human two-level memory system. The resulting *component fault rate model* accurately predicts the observed data for all languages in this study.

Using this component fault rate model, the discussion follows on by building a simple *system fault rate model* and uses this model to make predictions about overall system fault rates. This system model suggests amongst other things, that *re-use* within an existing design may degrade it initially and also that systems built using medium sized components may be much more reliable than those built from larger components or smaller components. These conjectures need to be subjected to further experiments.

[1]Les Hatton is with Programming Research Ltd, Glenbrook House, Hersham, KT12 4RH, U.K. See also http://www.prqa.co.uk/

I. INTRODUCTION

In software engineering, the lack of experimental evidence often means that anecdotal, intuitive or sometimes just plain commercial arguments become surprisingly well entrenched. For example, in spite of the enormous and growing effort in object-orientated technology, there seems little if any, solid, repeatable evidence to prove that it delivers *any* of the benefits it promised on intuitive grounds, a situation which leaves me at least, somewhat incredulous. This is a relatively modern example but there are many such examples over the 40 year history of software engineering. In the 1980s, CASE provided similar extravagant claims which it was largely unable to deliver and database technology in the '60s and '70s went through many traumas before eventually realising most of its original promise. Knowledge based systems and formal methods have been similarly oversold and are just beginning to recover from the hype, with cautious progress. These are all symptoms of a relatively immature discipline and it should not perhaps be surprising if even fundamentally accepted principles are unsupported by any repeatable measurement.

One such principle which I subscribed to for many years is the belief that modularisation or structural decomposition is a good design concept and therefore always improves systems. This is so wide-spread as to be almost unchallengeable. It is responsible for the important programming language concept of compilation models which are either separate, with guaranteed interface consistency, (e.g. C++, 'new-style' C, Ada and Modula-2), or independent (e.g. 'old-style' C and Fortran), whereby a system can be built in pieces and glued together later. It is a very attractive concept with strong roots in the "divide and conquer" principle of traditional engineering. Of course, the proof of any such concept relies on substantiation by the observation and measurement of real systems. For example, in conventional engineering systems, the need for reliability suggests splitting an overall design into pieces. This makes sense for intellectual tractability, but it has long been known that *if designs are split into too many small pieces, reliability may be prejudiced*. This is embodied in the celebrated and highly pragmatic KISS engineering principle. By analysing the results of recent measurements of the reliability of software systems, this paper will show that the same applies in software and that modularisation may not always lead to more reliable systems.

(FOR EDITOR: Next 2 paragraphs could be side-bar on nomenclature).

Before discussing the data, a word or two on nomenclature is worthwhile. There is considerable confusion in the literature between the various use of error, fault, failure, mistake, defect and bug, when discussing reliability. This is compounded by the IEEE model of error -> fault -> failure when compared with the IEC 1508 safety-related software standard model of fault -> error -> failure.

In this paper, the IEEE model will be used and so fault will be used to denote an inconsistency in the code which may or may not lead directly to failure, depending on external circumstances.

To complicate matters further, fault density is the number of faults divided by some measure of size, usually source lines of code. Unfortunately, there is no standard definition of fault, nor of line of code. I have studied various sources which show a variation as high as 25:1 when severity of failure is included. In addition, in common languages, there is a variation of perhaps 2:1 in different definitions of lines of code. For example, in C, the total number of executable lines of code is approximately 50% of the total number of newlines on average. Both are commonly used measures. This makes comparing fault densities in different systems very difficult unless this is made explicit, although fortunately, studies tend to be self-consistent, so the phenomenon reported in this paper is not affected.

II. CASE HISTORIES

The central point of this paper revolves around measurements reported by [1], [2], [3], [4] and [5], all of whom reported the same phenomenon directly or indirectly, viz. *that small components tend to have a disproportionately **larger** number of bugs than bigger components*. Although this is highly suggestive, I was acutely aware that 5 studies do not make a water-tight case, but since then four more studies, [6], [7], and personal communications of Pascal data from Martin Shepperd and of C++ data from Watts Humphrey have come to my attention, all of which exhibit essentially the same phenomenon. In addition, there have been no conflicting studies. The above case studies span large and small developments in Ada, C, C++, Fortran, Pascal and various forms of assembly and macro-assembly language, from various parts of the world, and diverse application areas at different stages of maturity. In other words, they have little in common apart from the fact that each exhibits the same phenomenon. Note that in this sense, components represent an artificial interface in the program across which data-flow takes place. In many languages, this might correspond to a subroutine or function. In object-orientated languages such as C++, this might be a class or method.

My view was that this is far more than mere coincidence and it appears that software systems are exhibiting macroscopic behaviour in spite of their massive internal complexity, rather like the very large number of molecules in a gas nevertheless lead to the simple macroscopic relation PV = RT. Owing to their very disparate nature, the first five are discussed in the order in which I became aware of them. The remaining four corroborate the first five, but will not be described in detail.

A. Hatton and Hopkins 1989

This paper studied the internationally famous NAG Fortran scientific subroutine library, comprising some 1600 routines totalling around 250,000 executable lines. The NAG library is very attractive to the software experimentalist because it has been through some 15 separate releases in the more than twenty years since its first appearance, and has a complete bug and maintenance history embedded in machine extractable form in the header of each component routine. Amongst other things, we found that the number of bugs in the library was well-predicted by the formula:

$$N_{bugs} = \mu \log_{10} (\rho . \Omega) \qquad (1)$$

where μ and ρ are scalars and Ω is a measure of the complexity, in this case the static path count, [8]. At the time, we were quite happy with the notion that more complex components had more bugs, but we had not thought the implications through in that *it follows immediately from its less than linear relationship, that smaller components contain proportionately more bugs than larger components*.

It is worth noting also that the NAG library is fundamentally different from the other systems in this paper. It is a library of re-usable components and does not of itself constitute a system. That it shares the same behaviour as components bound together in the same system, further supports the view that the behaviour described in this paper is a macroscopic property.

B. Davey, Huxford et al. 1993

Some 4 years later, following a detailed analysis of a well-measured but very different engineering project, i.e. a different programming language - C, and a different part of the life-cycle - development, these authors reported precisely the same phenomenon. Once again, it was the smaller components which contained proportionately more faults. In this case however, the measure of complexity used was a count of the source code lines.

C. Moller 1993

In the same year, Moller, in a detailed analysis of an entirely different type of development, this time several versions of operating systems written primarily in various dialects of assembler, again reported the same phenomenon, that the smaller components were proportionately more unreliable. Again, complexity was measured in terms of lines. However, this study contained some very large components also and these were found to grow much more quickly in unreliability again, restoring the intuitive view.

D. Compton and Withrow 1990

This paper yet again contrasts the other case studies reported in that it covers a large scale Ada development, a language supposedly free of many of the defects of other languages. Again, the authors found that small components were proportionately more unreliable. However, there were again some very large components in these authors' sample and like [5], they also found that the unreliability began to rise disproportionately again, leading to an optimum size at which unreliability in faults per KLOC (1000 lines of source code) was a minimum.

E. Basili and Perricone 1984

Although chronologically the first, the author became aware of this work rather late in the present study. This particular paper contained several important conclusions based on a study of a large suite of Fortran programs. From the point of view of this study, the most important conclusion was once again that small components were proportionately more unreliable than the larger components, a conclusion which troubled its authors to the point of trying to defend it. This is the earliest study so far seen by the author in which this effect has been observed and explicitly commented on.

F. A composite analysis

The strikingly similar qualitative behaviour of the above case histories coupled with the logarithmic behaviour observed by Hatton and Hopkins suggests using such a relationship to attempt to unify the data from the others quantitatively. As an example, the assembler data reported by [5] is plotted against a logarithmic curve as shown in Figure 1 below.

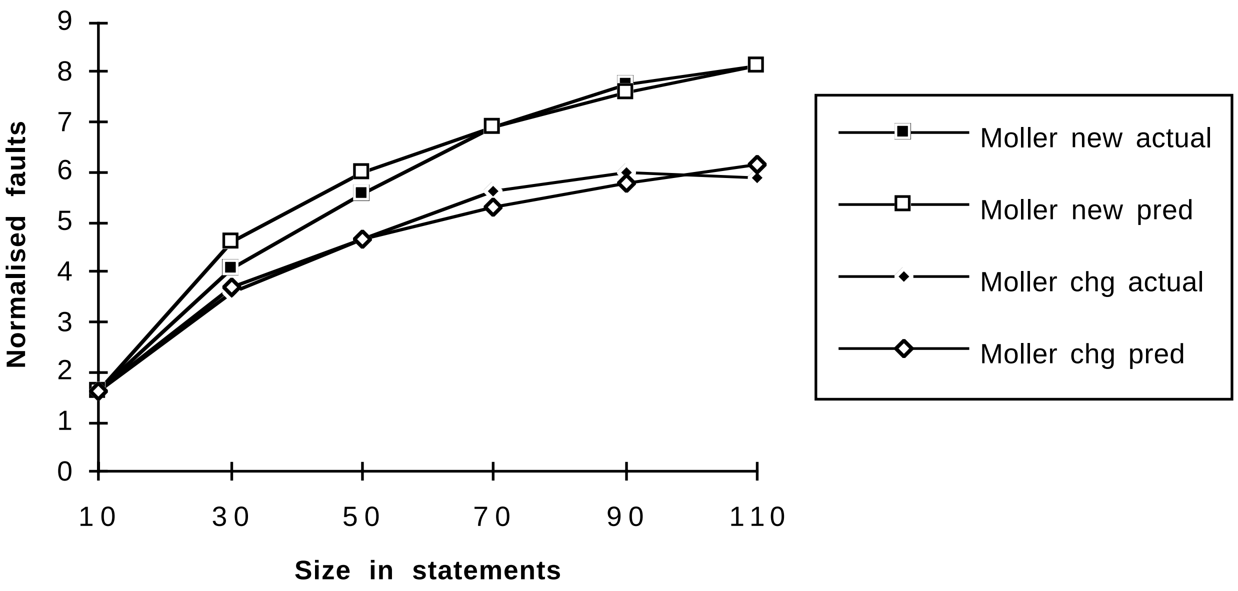

Figure 1: A comparison of the faults reported in Figs 6 and 7 of [5], compared with the predictions based on equation (1). The abbreviation *chg* stands for "changed" and data for both new and changed components are shown.

The other case studies show such qualitatively similar behaviour that I felt it worth while investigating if there is some underlying mechanism at work.. This is necessarily speculative but leads to a model from which interesting predictions can be made which could be tested by future experiments.

III. A Proposed Underlying Principle

It is easy to get the impression by studying the above case histories that developing software systems with low fault rates is exceedingly difficult. In fact, studying the available literature more widely, reveals graphs such as that shown in Figure 2. This data was compiled by the University of Maryland's Software Engineering Lab. from NASA Goddard data, as quoted in the December 1991 special edition of "Business Week". First of all, in spite of the enormous resources and pool of talent available, the average is still around 6 faults per 1000 lines. This ties in very closely with that reported in detailed studies by [9]. More telling is the observation that in Figure 2, improvement has been achieved mostly by improving the bad ones not the good ones suggesting that consistency, a process issue, has improved much more than actual fault density, a product issue. The Ada data of [4] also gives a figure of around 6 per 1000 lines. Along with other studies in the literature, the simple conclusion is that the average across many languages and development efforts for ‘good’ software is around 6 faults per 1000 lines, and that with the best techniques we know how to apply, 0.5-1 faults per 1000 lines can be achieved. Perfection will always elude of course, but the intractability of achieving better fault rates than have been achieved so far also suggests that a more powerful limitation may be at work.

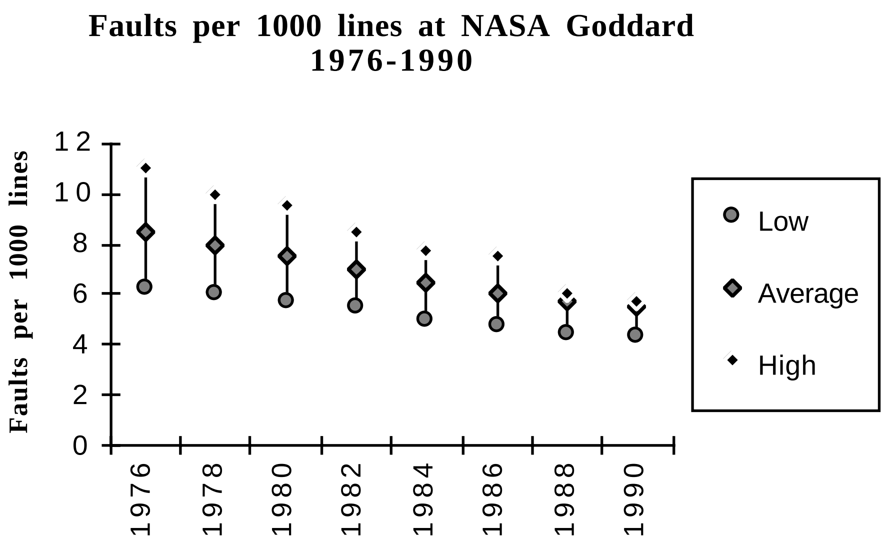

Figure 2: Faults per 1000 lines reported in an analysis of NASA Goddard data during the period 1976-1990.

A A discussion of faults per KLOC

The term faults per KLOC is frequently quoted as a measure of reliability in software systems. It is worthwhile exploring just what this means. At some point in time, a system starts to be used with hopefully no known faults in it. As time goes by and use increases, faults are encountered and reported. Hence faults per KLOC, the number of faults divided by some measure of the number of lines of code, is a function of time. In a system subject only to corrective maintenance, faults will eventually occur only rarely, so the time-dependent faults per KLOC will approach an asymptote as time increases. In reality, only this asymptote makes sense for comparing the reliability of different systems. So, given that the asymptote can never be reached, the faults per KLOC *and* the rate of change of this value are required to compare such systems effectively. Of course, real systems are subject to continual non-corrective change, so things become rather more complex. No notion of rate of change of faults per KLOC was available for any of the data in this study, although both mature and immature systems were present, *and still the same behaviour was observed*. This would suggest that the observed defect behaviour is present through the life-cycle, supporting even further the conjecture that it is a macroscopic property. If only immature systems had been present in the studies, it could have been argued that smaller components may get exercised more. This does not seem to be the case.

A further related point which was also observed in the study of the NAG library by [2], is that when component fault rates are plotted as a function of size, the usage of each component must be taken into account. The models discussed later in this paper are essentially asymptotic, and the fault rates they predict are therefore an envelope to which component fault rates will tend only as they are used sufficiently to begin to flush out faults. An unused component has complexity but has no faults by definition. I am aware of apparently near zero-defect systems in the literature which have turned out on closer inspection to have been unused.

The rest of this paper should be read with these observations in mind.

B A physiological model for component fault rates

Given the ubiquitous nature of the behaviour described earlier, it is worthwhile pursuing a possible model for it. At the very least, such a model would have to explain not only the logarithmic behaviour of faults with component size for small to medium components, but also the rapid and qualitatively similar departure from this behaviour for very large components reported by [4] and [5], for *totally different languages*. This latter factor gives a strong clue that the cause may be associated with the way the human mind manipulates symbolic data, rather than any specific property of the software itself, as software is no more than a frozen thought.

Whilst simple probabilistic models can be constructed, these usually degenerate into mere data fitting. However, the following pieces of work amongst others, all provide hints as to the underlying mechanism used in the brain during the act of comprehension and symbolic manipulation:

(a) The seminal work of Miller, [10] shows that humans can cope with around 7 ± 2 pieces of information at a time through the mechanism of the short-term memory. *Note that this appears to be independent of the information content of the pieces.* For example a binary sequence contains inherently less information than a sequence of denary numbers but the length of the sequence that can be absorbed and manipulated is around the same. This is strongly evocative of the language independent component fault rate behaviour described earlier.

Miller also described the notion of *chunking* by which a problem is systematically broken down into chunks which fit into short-term memory during the understanding process.

(b) Hilgard and co-workers, [11], argue that the short-term memory incorporates a *rehearsal buffer* which continuously refreshes its contents. These authors also describe the standard memory model whereby a long-term memory backs up the short term memory but acts in a fundamentally different way in that its contents are in a coded form and to all intents and purposes are never lost even though the recovery codes may get scrambled in various conditions. There is very considerable psychological and physiological evidence to support this model, from the studies of Alzheimer's disease for example, which affects only the short-term memory. The scrambling of the recovery codes will be an important factor in the model which follows.

Putting together the above evidence suggests a form of short-term or *cache* memory which makes the understanding of anything which fits considerably easier and less fault-prone than something which does not fit which overflows, risking scrambling of the recovery codes used for long-term storage.

If a programmer is working with a component of complexity Ω, which entirely fits into the cache or short-term memory which can be manipulated without recourse to back-up or long-term memory, the incremental increase in bugs or disorder dE due to an incremental increase of complexity of d Ω is simply:

$$dE = \frac{1}{\Omega} d\Omega \tag{3}$$

This follows along a similar argument to that leading to Boltzmann's law relating entropy to complexity, where the analogue of equipartition of energy in a physical system is mirrored by the apparently equal distribution of rehearsal activity in the neural complex comprising the short-term memory. In other words, because no part of the cache is favoured and it is very accurate in symbolic manipulation, the incremental increase in disorder is inversely proportional to the complexity which already exists, making the ideal case when pieces just fit into cache. It is assumed without loss of generality that both E and Ω are continuously valued variables.

What happens when complexity greater than Ω ', that complexity which will just fit into the cache, is encountered ?

In this case, we can see that the increase in disorder will correspond to the complexity in the (now-full) cache contents plus a contribution proportional to the number of times the cache memory has to be reloaded from the long-term memory. In other words:

$$dE = \frac{1}{2\Omega'}(1 + \frac{\Omega}{\Omega'})d\Omega \qquad (4)$$

The factor of (1/2) matches equation (3) when $\Omega = \Omega'$, i.e. when the complexity of the program is about to overflow the cache memory. The second term is directly proportional to the cache overflow effect and mimics the scrambling of the recovery codes.

Integrating equations (3) and (4) suggests that:

$$E = \log(\Omega) \text{ for } \Omega \leq \Omega' \qquad (5)$$

and

$$E = \frac{1}{2}(\frac{\Omega}{\Omega'} + \frac{1}{2}.\frac{\Omega^2}{\Omega'^2}) \text{ for } \Omega > \Omega' \qquad (6)$$

It is noteworthy that the logarithmic behaviour observed for small to medium-sized components in real systems naturally emerges from this argument. Whether or not the quadratic behaviour implied by (6) also emerges, will now be tested against the two earlier datasets which contain components varying from small to medium through to the very large.

As can be seen, the Ada data of [4] and the assembler and macro-assembler data of [5] both provide strong empirical support for this behaviour, with about 200-400 lines corresponding to the complexity Ω' at which the short-term or cache memory overflows into the long-term memory. That such disparate languages can produce approximately the same transition point from logarithmic to quadratic behaviour supports the view that Ω is not the underlying algorithmic complexity but the *symbolic* complexity of the language implementation, given that a line of Ada would be expected to generate 5 or more lines of assembler. This is directly analogous to the observation by [10] that the actual information content of the cache is irrelevant, simply that its *encoding* fits into cache.

Figure 3 plots the fault data of [4] and [5] along with a prediction using the model given by equations (5) to (6) above, assuming a cache overflow value of $\Omega' = 200$ lines of code.

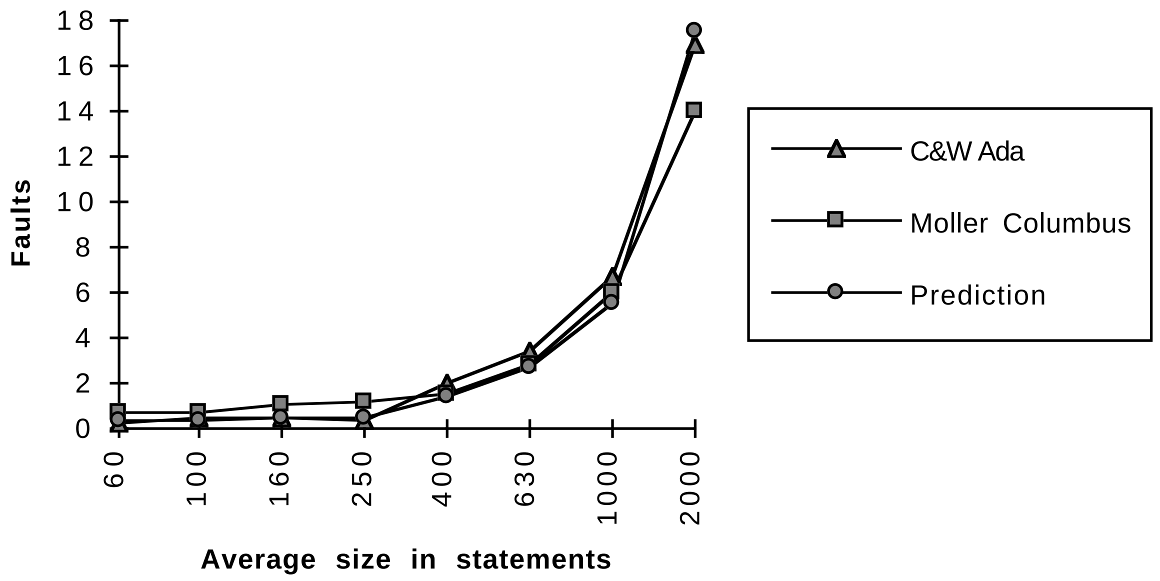

Figure 3. Ada fault data from [4] and Columbus assembler fault data from [5] plotted against the predictions of equations (5) and (6) of the text. Equation (5) is used to predict up to around 200 lines at which the short-term cache memory is assumed to overflow with the level of complexity. From then on, equation (6) is used to predict the fault growth. The quality of agreement gives strong empirical support for the model described in the text.

It should be noted at this point that if this behaviour is shown in terms of fault density, a U-shaped fault density against size curve results as pointed out for example by [4]. Plotting the data of Figure 3 in such a way gives the diagram of Figure 4. This figure also suggests that smaller components in Ada are more robust than smaller components in assembler but larger components are comparable. Although this comprises only one comparison, the author cannot resist speculating that the effects of 20 years of language sophistication may manifest themselves here in producing better smaller components whilst leaving larger components and the fundamental U-shape unaffected. This further supports the central thesis of this paper that this is because these latter two phenomena are not linguistically related.

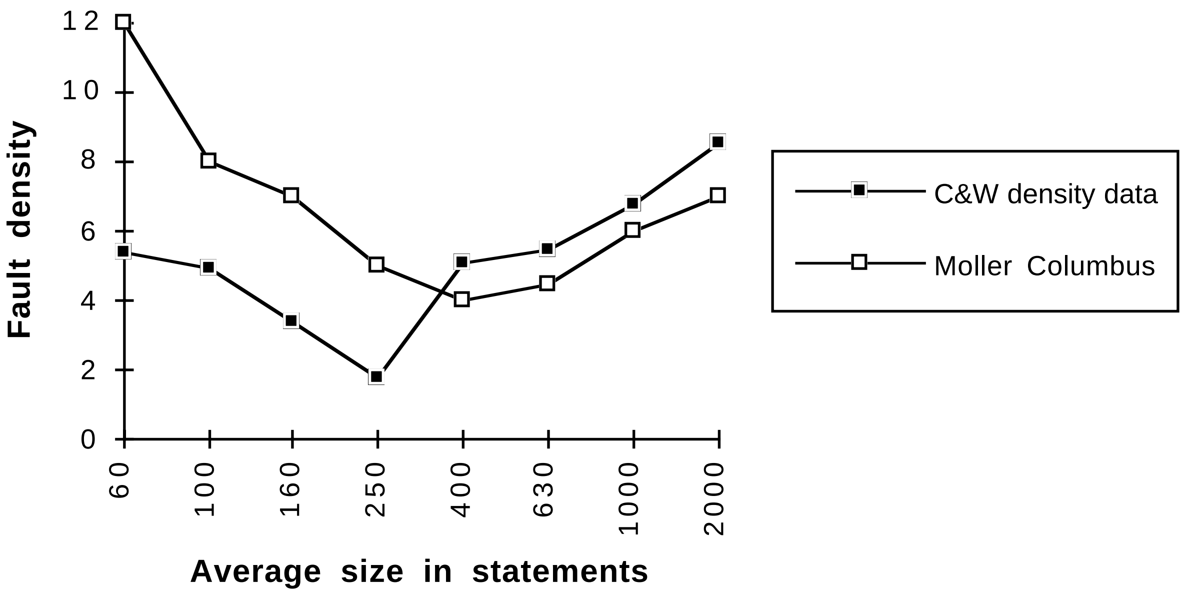

Figure 4: The data of Figure 5 plotted in terms of fault density, exhibiting a U-shaped curve for both the Ada and assembler data.

To attempt to summarise the behaviour described in this section in basic terms, if a system is decomposed into pieces much smaller than the short-term memory cache, the cache is used inefficiently because the interface of such a component with its neighbours is not 'rehearsed' explicitly into the cache in the same way, and the resulting components tend to exhibit higher defect rates. If components exceed the cache size, they suffer deterioration of comprehension because of the break down of the recovery codes connecting comprehension with long-term memory. Only those components which just fit into cache use it effectively, thereby producing the lowest fault densities.

IV. Implications For Software Development

The properties of the above model can be used to provide a potential explanation for other observations in software development generally.

A Large variations in individual programmer performance

Large variations in individual programmer performance have often been reported. Supposing that one of the manifestations of this is the known 7±2 variation in the 'size' of short-term memory reported by [10]. This would cause the change-over point between logarithmic and quadratic behaviour to vary by as much as 80% (5 - 9) in a typical population of programmers. This would cause quadratic behaviour to onset much earlier with some programmers than others. Simple modelling suggests that this could lead to a variation of a factor of 2-3 in the reliability of components produced by different programmers. Anecdotal evidence suggests that the real difference may be rather more than this, so other factors such as education may also be an issue. This hypothesis could be tested by experiment, for example by comparing defect rates in small components which should not be so badly affected.

B Why Inspections are very effective

In a manual code inspection, multiple independent short-term memories are being used to 'execute' the code statically. In other words, a code inspection is like a static N-version experiment. Even allowing for the known non-independence of such experiments, a considerable improvement still accrues, which may well explain the frequently reported effectiveness of manual inspection. The model represented here suggests that this would be most effective on components which fit into cache. Manual code inspections of components well into the quadratic zone should not be expected to improve things much since it will be beyond the easy grasp of any inspector to understand. Again, this hypothesis could be tested by experiment.

V. A SIMPLE SYSTEM BEHAVIOUR MODEL

So far, only component defect densities have been described and modelled. However, in order to make conjectures about the behaviour of a system as a whole, a speculative model with a simple component size distribution will be described to predict the likely number of faults in a *system* using the observed behaviour of faults against complexity for *components*. Although the model is very simple, the implications of simple logarithmic behaviour for small to medium size components are quite profound. To set the scene, consider the following argument for the design of a new system. This is very simplistic but retains essential features.

Suppose, that a particular functionality requires 1000 "lines" to implement, where a "line" is some measure of complexity. The immediate implication of the earlier discussion is that, on reliability grounds, it is far better to implement it as 5 x 200 line components (i.e. each fitting in cache) rather than 50 x 20 line components for example. The former would lead to perhaps 5 $\log_{10}(200) = 25$ bugs whilst the latter would lead to 50 x $\log_{10}(20) = 150$ bugs. This apparently inescapable but unpleasant conclusion runs completely counter to conventional wisdom, although the intuitive viewpoint might be restored by some combination of the following three mitigating factors:

- If splitting the system up into small components reduced the number of lines necessary to implement the required functionality, by the mechanism of *re-use*. However, a small calculation reveals that the reduction in size would have to be dramatic indeed. In the above example, an 80% reduction in size or so would be necessary. In practice, values considerably less than this are reported, e.g. [12]..
- If the additional unreliability due to splitting up the system into small components is due to simple interface inconsistencies. This was considered as a possible explanation in [1]. However, although this is considered to be important in [5], it was not a factor in [2] since the NAG library is a set of largely externally used re-usable components, and was found to have high interface consistency in the internally re-usable components. Furthermore, this is unlikely to explain the data recorded by [4] as Ada, the language of that study, mandates interface consistency in language implementations.
- It may be that the overall maintenance cost is reduced by modularisation even though the corrective component is higher. However, studies such as [13] report a corrective contribution of around 50% of all maintenance, so again this explanation may not be valid. Note also that whatever the corrective maintenance contribution may be, this issue has particular relevance for safety-critical systems, where reliability would generally be much more important than ease of change.

So, even this very simple model has profound implications. Now define the overall complexity Ω, of a software system consisting of N small to medium components each of implementation complexity Ω_i by:

$$\Omega = \prod_{i=1}^{N} \Omega_i + f(\Omega_i) \tag{7}$$

Here, the function $f(\Omega_i)$, is an unknown function depending on other combinations of the complexity of the individual components, and the product has been defined because the logarithmic component fault rate behaviour described earlier falls out nicely. In general, for components of small to medium size, it will be assumed for now that the first term on the right hand side dominates the second term.

Taking the logarithm of (7) gives:

$$E_t = \log\Omega = \log\left\{\prod_{i=1}^{N}\Omega_i\left[1 + \frac{f(\Omega_i)}{\prod_{i=1}^{N}\Omega_i}\right]\right\} \tag{8}$$

where E_t is the total number of faults. Now for small to medium components, using the assumption that

$$\left|\frac{f(\Omega_i)}{\prod_{i=1}^{N}\Omega_i}\right| << 1 \tag{9}$$

Equation (8) becomes:

$$E_t = \log\Omega = \log\prod_{i=1}^{N}\Omega_i + \frac{f(\Omega_i)}{\prod_{i=1}^{N}\Omega_i} \tag{10}$$

The reason for this rather contrived model is that equation (10) nicely embodies the logarithmic behaviour of equation (3) as well as the important observation by [1] , that most of the faults (89% in their case), in a real system affected only a single component, and therefore the total faults in the system was approximately the sum of the component faults, neglecting the second term in (10). This model has precisely such behaviour in that the total faults in a system can be approximated by

$$E_t = \log\Omega = \log\prod_{i=1}^{N}\Omega_i = \sum_{i=1}^{N}\log\Omega_i \tag{11}$$

It is from this model that comparisons with real systems will be described and some further predictions made.

First, assume that a system with E faults in total is made up of N components of equal size with a total number of lines, L, and that the number of lines in each component is used as a measure of its complexity. Although not essential, it will be further assumed that E is an asymptotic value for a stable system in the sense discussed earlier. Then from equation (11),

$$E = N \log \frac{L}{N} \tag{12}$$

Using equation (12) and defining E_{LOC} to be the number of system faults per line of code yields

$$E_{LOC} \equiv \frac{E}{L} = \frac{N}{L} \log \frac{L}{N} \tag{13}$$

From which E_{KLOC}, the number of system faults per 1000 lines of code as conventionally used is given simply by

$$E_{KLOC} = 1000 E_{LOC} \tag{14}$$

Various properties of equations (13-14) will now be studied.

A. Comparing systems with different average component size

The first property to note of equation (13) is that it has a *maximum*, as shown in Figure 5. The maximum occurs for very small objects of a few lines only. The implication of this is that object-orientation may make things *less* reliable in terms of total system fault unless the objects are very small indeed. This prediction may however be an artefact only and will have to be tested by experiment as the case histories reported earlier provide no data in this region and the approximation used going from equation (10) to (11) may be invalid. Away from this region, overall system reliability can be expected to improve inexorably as average component size increases. This argument will break down for component sizes beyond 200-400 lines as discussed earlier, where individual component logarithmic behaviour breaks down and the quadratic behaviour observed in the systems studied here, takes hold.

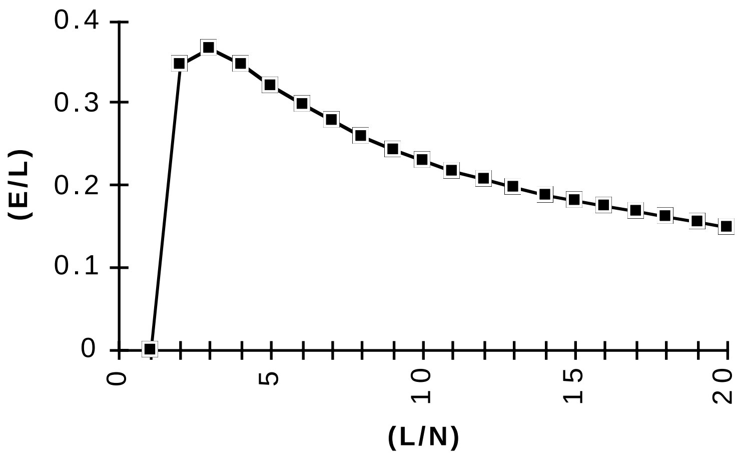

Figure 5: Graph of equation (13) of the text showing the maximum in faults per line (E/L). The x axis is the average component size in lines, (L/N).

B. Total system complexity

The argument which led up to equation (11) suggests that total system complexity can approximately be modelled as the product of the original component complexities as

$$\Omega = \prod_{i=1}^{N} \Omega_i \tag{15}$$

C. Changing an existing system

For an existing system of L lines and N components, differentiating equation (12) gives

$$\frac{\partial E}{\partial L} = \frac{N}{L} \tag{16}$$

and

$$\frac{\partial E}{\partial N} = \log\left(\frac{L}{N}\right) - 1 \tag{17}$$

This leads to three possibilities for improving the reliability of an existing system, i.e. reducing E.

(a) $(\partial E / \partial L) > 0$, keeping N constant and L **de**creasing.

This corresponds simply to reducing the total number of lines for the same number of components. This is essentially the mechanism of re-use.

(b) $(\partial E / \partial N) > 0$, keeping L constant and N **de**creasing.

This corresponds to reducing the number of components for the same number of lines. This simply increases the trend to large monolithic components.

(c) $(\partial E / \partial N) < 0$, keeping L constant, with N **in**creasing.

This case only holds for components which are very small already. It suggests that a trend to small components via the mechanism of object-orientation for example, will improve overall system reliability *only* if existing components are already very small.

The benefit of change can be estimated by studying Figure 5. Systems with small components around the maximum of that curve should benefit the most, either by making them smaller still or by making them larger, because the gradient of improvement is steepest nearest the maximum. On the other hand, in a system with relatively large components on average, little benefit will accrue by making them yet larger as the curve is much flatter here.

VI. CONCLUSION

This paper brings together compelling empirical evidence from very disparate sources for the thesis that in any software system, *larger components are proportionately more reliable than smaller components.* This contradicts conventional wisdom as evidenced by the number of authors who were surprised at the results they found, (including myself !).

Given that this behaviour spans multiple languages, (both object-orientated and other), mature and immature systems and also non tightly-coupled systems, (e.g. the NAG library), as well as tightly coupled systems, (operating systems), I would conjecture that this may well be the first quantitative indications that macroscopic fault behaviour is present in software systems and that there may be limitations on the fault rate which can be achieved. It is also raises the possibility of predicting certain properties of software systems from basic static parameters like component size distribution.

The paper then constructs a model of *component fault rate* as a function of size using known physiological properties of the human memory system. This component model predicts

- **logarithmic** behaviour up to the capacity of the short-term memory (in the region of 200-400 lines and apparently independent of language)
- **quadratic** behaviour for component sizes of larger than this cut-off value, as program complexity causes a spill-over into the long-term memory.

The model closely predicts the observed qualitative behaviour in the real systems discussed here. The paper is not however able to quantify complexity in this sense, and further work is necessary here.

Finally, the paper uses the above model of *component* fault rate behaviour to predict *system* fault rate behaviour for systems with very simple distributions of component size. This system model *predicts*, amongst other things:

- There may be a maximum fault-rate (for very small components) as well as the minimum fault-rate (for rather larger components) suggested by some of the case histories. This is very tentative evidence that object-orientation may deliver increased reliability but only if the components are very small indeed, i.e. of the order of 1 or 2 lines. There is as yet no evidence to support or disprove this conjecture, although defect density rates for functional languages may be able to shed light on this.
- Multiplying the complexity of each component is a reasonable measure of overall system complexity provided the components are not too large.
- Only very substantial re-use within the same system is likely to improve the reliability of a given system. Modest re-use within the same system is likely to make the system worse. This paper postpones for now the discussion of re-use of old components in a new system.
- When changing an existing system, the direction of increasing reliability depends on the existing average component size.
- The most reliable systems may result from systems with component sizes grouped around the 200-400 line mark. Bigger and smaller average component sizes appear to degrade this reliability.

Further experiments and analysis will be necessary to support or disprove these conjectures. To re-iterate however, there is nothing conjectural about the fact that published reliability studies are currently in serious conflict with the conventional wisdom that structural decomposition or modularisation of systems into small, easily manageable components makes better systems. In terms of reliability, it almost certainly does not, even if they become easier to change as a result, and even this is unproven. This has serious implications for high-integrity software development. The apparent existence of a trade-off between changeability and reliability needs to be studied further.

It is intriguing to speculate whether this phenomenon pervades other areas of human creativity, but this is well beyond the scope of this paper.

ACKNOWLEDGEMENT

The author would like to thank his colleagues at Programming Research Ltd. for numerous discussions on this topic. He would also like to thank Tom Anderson for stimulating discussions about models for fault behaviour and Norman Fenton for bringing to his attention some of the above case histories, which were instrumental in building the argument. Unknown reviewers made several important suggestions for improvements as did Tony Hoare who read an early version of these arguments and contributed significantly, offering further insights, much advice and constructive criticism. Finally, I would like to thank the authors of the published data without whose efforts no systematic progress can be made.

REFERENCES

1. Basili, V.R. and B.T. Perricone, *Software Errors and Complexity: An Empirical Investigation.* Comm. A.C.M., 1984. : p. 42-52.

2. Hatton, L. and T.R. Hopkins. *Experiences with Flint, a software metrication tool for Fortran 77*. in *Symposium on Software Tools*. 1989. Napier Polytechnic, Edinburgh:

3. Davey, S., *et al.*, *Metrics Collection in Code and Unit Test as part of Continuous Quality Improvement.* Journal of Software Testing, Verification and Reliability., 1993. **3**: p. 125-148.

4. Compton, B.T. and C. Withrow. *Improving Productivity: Using Metrics to Predict and Control Defects in Ada Software*. in *Second Annual Oregon Workshop on Software Metrics*. 1990. Oregon:

5. Moller, K.-H. and D.J. Paulish. *An empirical investigation of software fault distribution*. in *CSR'93*. 1993. Amsterdam: Chapman-Hall.

6. Shen, V.Y., *et al.*, *Identifying error-prone software - an empirical study.* IEEE Transactions on Software Engineering, 1985. **SE-11**(4): p. p. 317-323.

7. Kitchenham, B. and P. Mellor, *Data collection and analysis*, in *Software Metrics: a rigorous approach.*, N.E. Fenton, Editor^Editors. 1991, Chapman & Hall: London. p. p. 89-110.

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

9. Musa, J., Iannino., and Okumuto., *Software Reliability: Measurement, Prediction, Application*. 1987, McGraw-Hill.

10. Miller, G.A., *The magical number 7 plus or minus two: Some limits on our capacity for processing information.* Psychological Review, 1957. **63**: p. 81-97.

11. Hilgard, E.R., R.C. Atkinson, and R.L. Atkinson, *Introduction to Psychology*. Fifth ed. 1971, New York: Harcourt Brace Jovanovich. 640.

12. Frakes, W.B. and C.J. Fox, *Sixteen questions on software re-use.* Comm. A.C.M., 1995. **38**(6): p. p. 75-87.

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

FIGURE CAPTIONS

Figure 1: A comparison of the faults reported by [4] and [1] compared with the predictions based on equation (1). There is a missing point at 90 statements in the Basili & Perricone data.

Figure 2: A comparison of the faults reported in Figs 6 and 7 of [5] , compared with the predictions based on equation (1). The abbreviation *chg* stands for "changed" and data for both new and changed components are shown.

Figure 3: A comparison of the faults reported by [4] compared with the predictions based on equation (1). The curve appears to be dipping down to the right but starts rising again in the actual data. It has been plotted to this rather truncated scale to compare more easily against Figures 1 and 2.

Figure 4: Faults per 1000 lines reported in an analysis of NASA Goddard data during the period 1976-1990.

Figure 5. Ada fault data from [4] and Columbus assembler fault data from [5] plotted against the predictions of equations (5) and (6) of the text. Equation (5) is used to predict up to around 200 lines at which the short-term cache memory is assumed to overflow with the level of complexity. From then on, equation (6) is used to predict the fault growth. The quality of agreement gives strong empirical support for the model described in the text.

Figure 6: The data of Figure 5 plotted in terms of fault density, exhibiting a U-shaped curve for both the Ada and assembler data.

Figure 7: Graph of equation (13) of the text showing the maximum in faults per line (E/L). The x axis is the average component size in lines, (L/N).

INDEX ITEMS

Reliability

Re-use

Language-independence

Logarithmic behaviour

Systems and components

PREFERRED ADDRESS FOR CORRESPONDENCE

Les Hatton
"Oakwood"
11, Carlton Road
New Malden
Surrey, KT3 3AJ
U.K.

1. Basili, V.R. and B.T. Perricone, *Software Errors and Complexity: An Empirical Investigation.* Comm. A.C.M., 1984. : p. 42-52.

2. Hatton, L. and T.R. Hopkins. *Experiences with Flint, a software metrication tool for Fortran 77*. in *Symposium on Software Tools*. 1989. Napier Polytechnic, Edinburgh:

3. Davey, S., *et al.*, *Metrics Collection in Code and Unit Test as part of Continuous Quality Improvement.* Journal of Software Testing, Verification and Reliability., 1993. **3**: p. 125-148.

4. Compton, B.T. and C. Withrow. *Improving Productivity: Using Metrics to Predict and Control Defects in Ada Software*. in *Second Annual Oregon Workshop on Software Metrics*. 1990. Oregon:

5. Moller, K.-H. and D.J. Paulish. *An empirical investigation of software fault distribution*. in *CSR'93*. 1993. Amsterdam: Chapman-Hall.

6. Shen, V.Y., *et al.*, *Identifying error-prone software - an empirical study.* IEEE Transactions on Software Engineering, 1985. **SE-11**(4): p. p. 317-323.

7. Kitchenham, B. and P. Mellor, *Data collection and analysis,* in *Software Metrics: a rigorous approach.,* N.E. Fenton, Editor. 1991, Chapman & Hall: London. p. p. 89-110.

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

9. Musa, J., Iannino., and Okumuto., *Software Reliability: Measurement, Prediction, Application.* 1987, McGraw-Hill.

10. Miller, G.A., *The magical number 7 plus or minus two: Some limits on our capacity for processing information.* Psychological Review, 1957. **63**: p. 81-97.

11. Hilgard, E.R., R.C. Atkinson, and R.L. Atkinson, *Introduction to Psychology*. Fifth ed. 1971, New York: Harcourt Brace Jovanovich. 640.

12. Frakes, W.B. and C.J. Fox, *Sixteen questions on software re-use.* Comm. A.C.M., 1995. **38**(6): p. p. 75-87.

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