Reexamining the Fault Density–Component Size Connection

LES HATTON, Programming Research Ltd.

Conventional wisdom, that smaller components contain relatively fewer faults, may be wrong. This author found that medium-sized components were proportionately more reliable than small or large ones. Moreover, he says, there may be limits on the fault density we can achieve.

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 growth of object-oriented technology, there seems little if any solid, repeatable evidence that it delivers any of its promised benefits. This is a relatively modern example, but many such cases permeate the 40-year history of software engineering. In the 1980s, CASE provided similarly extravagant claims that it was largely unable to deliver, and database technology in the '60s and '70s went through many traumas before eventually realizing 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. It should not be surprising if even fundamentally accepted principles are unsupported by repeatable measurement.

0740-7459/97/$10.00 © 1997 IEEE

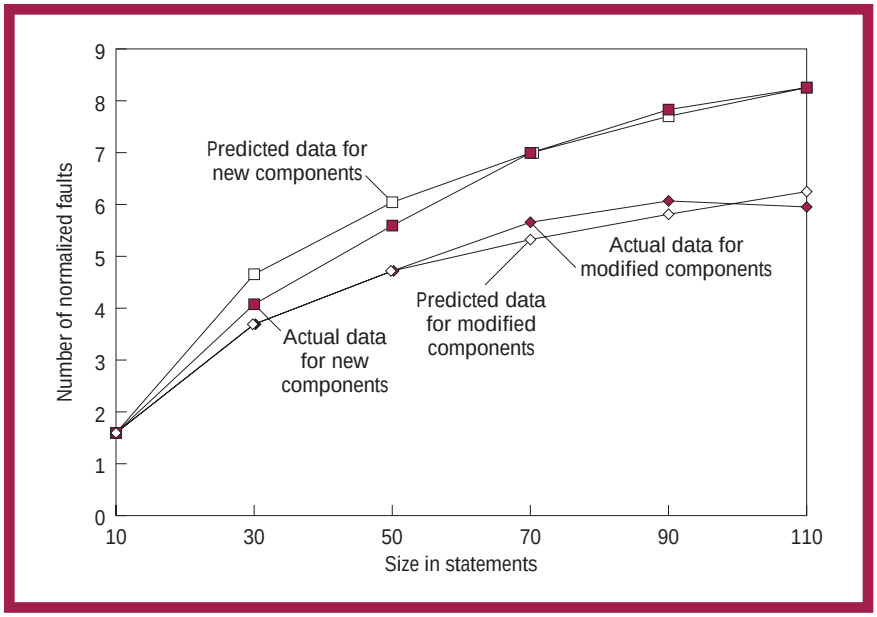

***Figure 1**. A comparison of the Moller–Paulish fault data with predictions based on Equation 1. Data for both new and modified components is shown.*

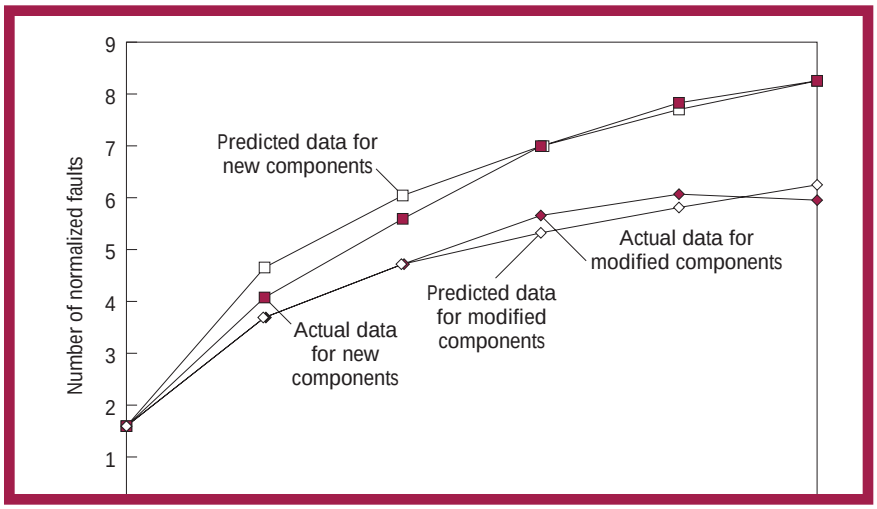

***Figure 2.** Faults per 1,000 lines of code, reported in an analysis of 1976–1990 NASA Goddard data.*

For years I subscribed to such a principle: that modularization, or structural decomposition, is a good design concept and therefore always improves systems. This belief is so widespread 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 (such as C++, Ada, and Modula-2), or independent, whereby a system is built in pieces and glued together later (C and Fortran, for example). It is a very attractive concept with strong roots in the "divide and conquer" principle of traditional engineering. However, this conventional wisdom may be wrong. Only those components that fit best into human short-term memory cache seem to use it effectively, thereby producing the lowest fault densities. Bigger and smaller average component sizes appear to degrade reliability.

Of course, the proof of modularization or any such concept relies on substantiation, through observing and measuring real systems. For example, in conventional engineering systems, the need for reliability suggests splitting an overall design into pieces. This is embodied in the celebrated and highly pragmatic KISS ("keep it simple, stupid") engineering principle. This division makes design easy to manage, but if designs are split into too many small pieces, reliability may be prejudiced. By analyzing the results of some reliability studies, I will show that the same applies in software: modularization does not always lead to more reliable systems. The boxed text on nomenclature, on page 91, might be helpful before we look at the data.

CASE HISTORIES

Directly or indirectly, several important studies have reported the same phenomenon—that small components tend to have disproportionately more bugs than bigger components.[1-7] My communications with Martin Shepperd about his Pascal data and with Watts Humphrey about his C++ data also support this conclusion. Although this does not add up to a watertight case, there have been no conflicting studies. The research spans 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 other than this one phenomenon. In this sense, components represent artificial, internal interfaces that data flows across. In many languages, this might correspond to a subroutine or function; in OO languages such as C++, this might be a class or method.

Is this more than coincidence? I think so. Software systems seem to exhibit macroscopic behavior in spite of their massive internal complexity, rather like countless molecules in a gas, which lead to the simple macroscopic relation $PV = RT$ (pressure × volume = the general gas constant × temperature). I examine five of the nine studies in the box on page 92. The remaining four corroborate the first five.

NOMENCLATURE

Discussions of reliability in the literature exhibit considerable confusion over the various uses of the terms *error*, *fault*, *failure*, *mistake*, *defect*, and *bug*. 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. This article uses the IEEE model, so *fault* will denote an inconsistency in the code that 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 lines of source code.

Unfortunately, there is no standard definition of fault, nor of line of code. Different sources show a ratio of benign to severe failure as high as 25:1. In addition, common languages vary as much as 2:1 in their definitions of lines of code. For example, on average, the total number of executable lines of C code is approximately half of the total number of newline characters. Both are commonly used measures. This makes comparing fault densities in different systems very difficult unless such definitions are made explicit. Fortunately, studies tend to be self-consistent, so the phenomenon reported here is not affected.

Faults per KLOC is often quoted as a measure of software system reliability. But what is it?

At some point in time, a system is implemented with the hope that it is fault-free. As time goes by and use increases, users encounter and report faults. 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 occur 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 noncorrective 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, with the same behavior observed. This would suggest that the observed defect behavior 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, also observed in the NAG library study, is that when component fault densities are plotted as a function of size, the usage of each component must be taken into account. The models discussed in this article are essentially asymptotic, and the fault densities they predict are therefore an envelope to which component fault densities will tend only as they are used sufficiently to begin to flush out faults. An unused component has complexity but no faults, by definition. The literature reports apparently near-zero-defect systems that have turned out on closer inspection to have been unused.

A composite analysis. The strikingly similar qualitative behavior of the case histories coupled with the logarithmic behavior that Hopkins and I observed suggest the use of such a logarithmic relationship when we try to quantitatively unify the data. As an example, Figure 1 shows the assembly data reported by Karl-Heinz Moller and D.J. Paulish[3] plotted against a logarithmic curve.

The other case studies showed similar behavior, prompting me to see if some underlying mechanism was at work. Although necessarily speculative, this study led me to devise a model from which we can make interesting predictions—predictions that can be tested in future experiments.

A PROPOSED UNDERLYING PRINCIPLE

It is easy to get the impression from these case histories that developing software systems with low fault densities is exceedingly difficult. In fact, analysis of the literature reveals graphs such as that shown in Figure 2. This data was compiled from NASA Goddard data by the University of Maryland's Software Engineering Laboratory, as quoted in the December 1991 special edition of *Business Week*. First of all, in spite of NASA's enormous resources and talent pool, the average was still five to six faults per KLOC. Other studies have reported similar fault densities.[4,8] More telling is the observation that in Figure 2, improvement has been achieved mostly by improving the bad processes, not the good ones. This fact suggests that consistency, a process issue, has improved much more than actual fault density, a product issue. The simple conclusion is that the average across many languages and development efforts for "good" software is around six faults per KLOC, and that with our best techniques, we can achieve 0.5–1 fault per KLOC. Perfection will always elude us, of course, but the intractability of achieving systematically better fault densities than have been achieved so far also suggests that some other limitation may be at work.

Given the ubiquitous nature of this trend, it is worthwhile to attempt to model it. At the very least, this model would have to explain not only the logarithmic relationship of faults to component size for small to medium components, but also the rapid but qualitatively similar departure from this behavior for very large components, for different languages. This latter factor suggests that the cause may relate to the way the human mind manipulates symbolic data, rather than to any specific property of the software itself.

Although simple probabilistic models can be constructed, these usually degenerate into mere data fitting. Instead, I chose to base my model upon findings from human memory research.

The seminal work of G. Miller shows that humans can cope with around 7 ± 2 pieces of information at a time via short-term memory, independent of information content.[9] For example, a binary sequence contains inherently less information than a sequence of base-10 numbers, but the length of the sequence that can be

THE EVIDENCE

Tim Hopkins and I studied[1] the internationally famous NAG Fortran scientific subroutine library, comprising some 1,600 routines totalling around 250,000 executable lines. The NAG library appeals to the software experimentalist because it has been through 15 releases over more than 20 years, and because it has a complete bug and maintenance history embedded in machine-extractable form in each component routine's header. Among other things, we found that the number of bugs in the library was well-predicted by the formula

$$N_{bugs} = \mu \log_{10} (\rho \times \Omega),$$

where μ and ρ are scalars and Ω is a measure of the complexity, in this case the static-path count.[13] At the time, we were quite happy with the notion that more complex components had more bugs. However, we had not thought the implications through; it follows immediately from its less-than-linear relationship that smaller components contain proportionately more bugs than larger components.

Note that the NAG library is fundamentally different from the other systems described below. It is a library of reusable components, not a system in itself. The fact that it shares the same behavior as components connected within a system further supports the view that the behavior I describe is a macroscopic property.

Some four years later, following a detailed analysis of a well-measured but very different engineering project (using a different programming language, C, in a different part of the development life cycle), S. Davey and colleagues reported precisely the same phenomenon: smaller components contained proportionately more faults.[2] In this case, however, the complexity measure was a count of source code lines.

In a detailed analysis of an entirely different type of development—several versions of operating systems written primarily in various dialects of assembly—Karl-Heinz Moller and D.J. Paulish[3] reported the same phenomenon, again measuring complexity in terms of LOC. However, this study also contained some very large components whose unreliability grew much more quickly, restoring the intuitive view.

Research by B.T. Compton and C. Withrow[4] covered a large-scale development project in Ada, a language supposedly free of many of the defects of other languages. The sample contained small as well as very large components. Like Moller and Paulish, the authors found both that small components were proportionately more unreliable and that unreliability rose disproportionately as component size grew. The optimum size at which unreliability in faults per KLOC (1,000 lines of source code) was a minimum.

Vic Basili and B.T. Perricone[5] came to several important conclusions based on their study of a large suite of Fortran programs, the earliest study showing this phenomenon as far as I am aware. Their most important conclusion was, once again, that small components were proportionately more unreliable than the larger components, a conclusion which troubled its authors enough that they felt the need to discuss it in depth.

Finally, how small can a system be to exhibit this phenomenon? Recall that the *PV=RT* general gas equation holds down to very low vacuums indeed. Sitting on a train in Tokyo, I analyzed fault data on the GNU indent program kindly supplied by Rick Swanton. The application is only around 2,000 lines total, but once again the defect density curve is U-shaped with component size.

absorbed and manipulated is around the same. This is similar to the language-independent component fault density behavior described earlier. Miller also described the notion of chunking, by which a problem is systematically broken down into chunks that fit into short-term memory during the understanding process.

E.R. Hilgard and co-workers argue that the short-term memory incorporates a rehearsal buffer that continuously refreshes its contents.[10] They also describe the standard memory model whereby a long-term memory backs up the short-term memory but acts in a fundamentally different way—its contents are in a coded form and, to all intents and purposes, are never lost even though the recovery codes may get scrambled under various conditions. There is considerable psychological and physiological evidence to support this model, such as the studies of Alzheimer's disease, which show it affects only short-term memory.

THE PROPOSED MODEL

Recovery code scrambling is an important factor in my proposed model. The evidence suggests that anything that fits in a short-term or cache memory is easier to understand and less fault-prone; pieces that are too large overflow, involving use of the more error-prone recovery code mechanism used for long-term storage.

Thus, if a programmer is working with a component of complexity Ω, and that component fits entirely into the cache or short-term memory, which in turn 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 $d\Omega$ is simply

$$dE = (1/\Omega)\, d\Omega. \quad (1)$$

This resembles the argument 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 short-term memory. In other words, because no part of the cache is favored and the cache accurately manipulates symbols, the incremental increase in disorder is inversely proportional to the existing complexity, 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 we encounter complexity greater than Ω' (the complexity which will just fit into the cache)? The increase in disorder will correspond to the complexity in the (now-full) cache contents, plus a con-

tribution proportional to the number of times the cache memory must be reloaded from the long-term memory. In other words,

$$dE = \frac{1}{2\Omega'}\left(1 + \frac{\Omega}{\Omega'}\right)d\Omega. \quad (2)$$

The factor of 1/2 matches Equation 1 when $\Omega = \Omega'$, that is, 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 1 and 2 suggests that

$$E = \log \Omega \quad \text{for } \Omega \le \Omega' \quad (3)$$

and

$$E = \frac{1}{2}\left(\frac{\Omega}{\Omega'} + \frac{\Omega^2}{2\Omega'^2}\right) \text{ for } \Omega > \Omega'. \quad (4)$$

The logarithmic behavior observed for small to medium-sized components in actual systems emerges naturally from this argument. We can now test whether the quadratic behavior implied by Equation 4 also emerges from the two earlier data sets containing components of all sizes.

The Ada data and the assembly and macro-assembly data provide strong empirical support for this behavior, with about 200 to 400 lines corresponding to the complexity Ω' at which cache memory overflows into long-term memory. That such disparate languages can produce approximately the same transition point from logarithmic to quadratic behavior 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 five or more lines of assembly. This is directly analogous to the observation that it is fit, rather than the actual information content of the cache that is relevant.[9]

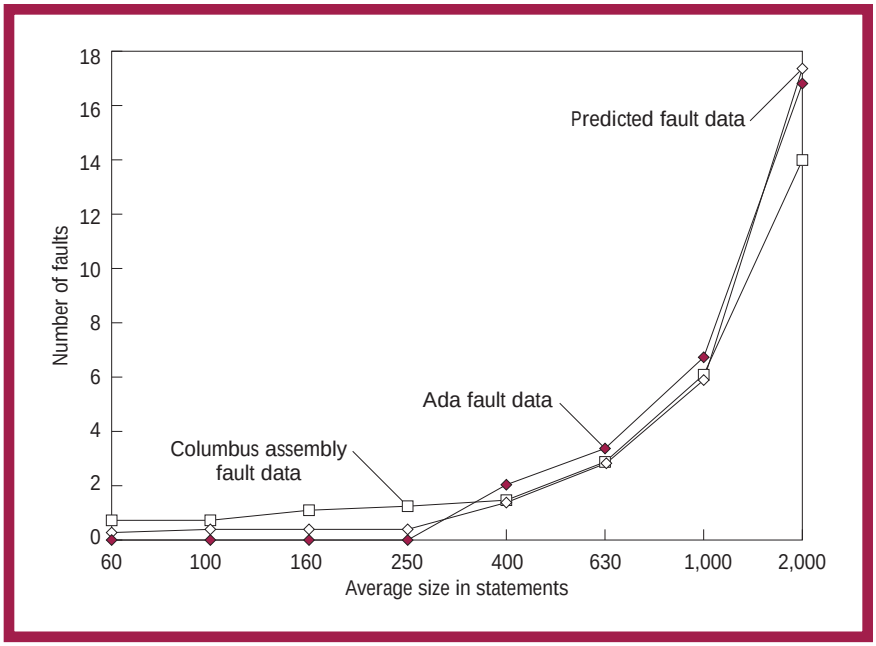

Figure 3. *Ada fault data and Columbus assembly fault data plotted against the predictions of Equations 4 and 5. I used Equation 4 to predict fault growth up to around 200 LOC, at which point the cache memory is assumed to overflow according to complexity level; from that point on, I used Equation 5. The quality of agreement gives strong empirical support for the proposed model.*

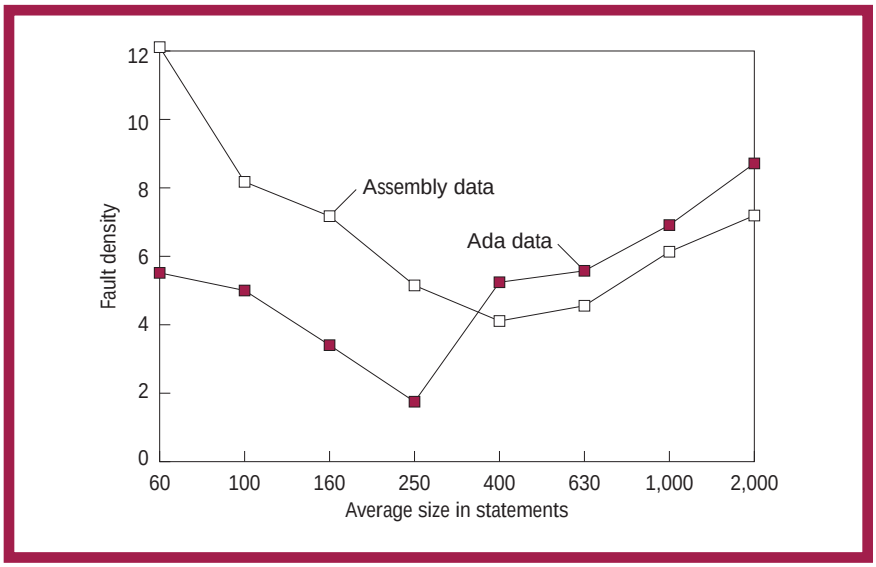

Figure 4. *The data of Figure 3 plotted in terms of fault density exhibits a U-shaped curve for both the Ada and assembly data.*

Figure 3 plots the fault data of two of the studies, along with a prediction using the model given by Equations 3 and 4, assuming a cache overflow value of $\Omega' = 200$ lines of code.[3,4]

If this behavior is shown in terms of fault density versus size, a U-shaped curve results.[4] Plotting the data of Figure 3 in such a way produces Figure 4. This figure also suggests that small Ada components are more robust than small assembly components but that larger components in the two languages are comparable. Although this is only one comparison, I cannot resist speculating that 20 years of language sophistication may produce better smaller components while leaving larger components and the fundamental U-shape unaffected.

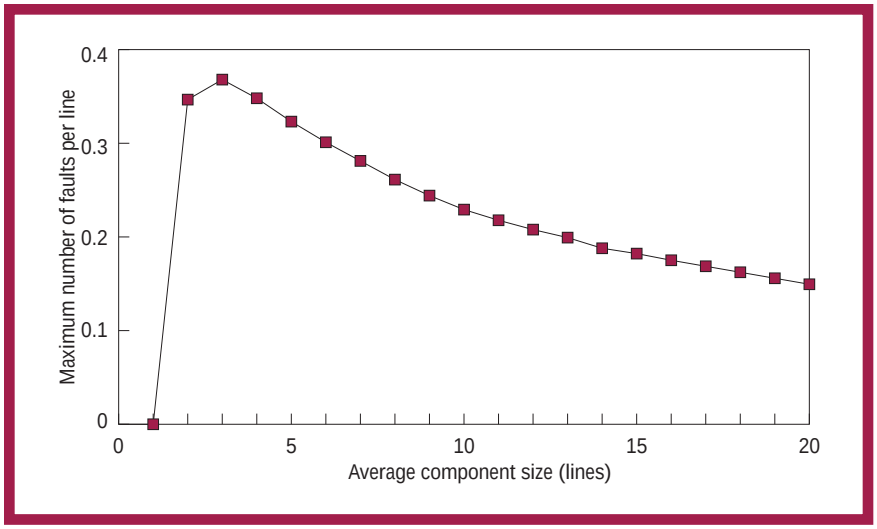

***Figure 5**. Graph of Equation 12 showing the maximum faults per line (E/L). The x axis is the average component size in lines, L/N.*

This further supports my central thesis that this is because the U-shaped phenomenon is not linguistically related.

To summarize, 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 neighbors is not "rehearsed" explicitly into the cache in the same way, and the resulting components tend to exhibit higher defect densities. If components exceed the cache size, they are less comprehensible because the recovery codes connecting comprehension with long-term memory break down. Only those components that match the cache size well use it effectively, thereby producing the lowest fault densities.

IMPLICATIONS FOR SOFTWARE DEVELOPMENT

This model may help explain two other things I've observed about software development. First, individual programmer performance varies widely. Suppose that one of the manifestations of this is the known 7 ± 2 variation in the "size" of short-term memory. This would cause the changeover point between logarithmic and quadratic behavior to vary by as much as 80 percent (5–9) in a typical programmer population; in other words, quadratic behavior would start much earlier with some programmers than with others. Simple modeling suggests this could lead to variation by a factor of 2 to 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. You could, for example compare defect densities in small components, which are less likely to be seriously affected.

The second observation is the effectiveness of inspection. In manual code inspection, multiple independent short-term memories are used to "execute" the code statically. In other words, a code inspection is like a static *N*-version experiment. Even allowing for the known nonindependence of such experiments, a considerable improvement still accrues, which may explain why manual inspection is often reported as effective. The model represented here suggests that manual inspection would be most effective on components that fit into cache. Manual code inspections of components well into the quadratic zone are unlikely to improve things much, because an inspector is unlikely to understand what's going on. Again, this hypothesis could be tested by experiment.

A SIMPLE SYSTEM BEHAVIOR MODEL

So far, I have described and modeled only component defect densities that were measured in the above case histories. However, to conjecture about system behavior as a whole, we must build a speculative model based on a simple component-size distribution to predict the likely number of system faults. Although the model is very basic, the implications of simple logarithmic behavior for small to medium-sized components are quite profound. To set the scene, consider the following simplified argument for the design of a new system.

Suppose that a particular functionality requires 1,000 "lines" to implement, where a "line" is some measure of complexity. The immediate implication of the earlier discussion is that, to be reliable, we should implement it as five 200-line components (each fitting in cache) rather than as 50 20-line components. The former would lead to perhaps 5 $\log_{10}(200) = 25$ bugs while the latter would lead to $50 \times \log_{10}(20) = 150$ bugs. This apparently inescapable but unpleasant conclusion runs completely counter to conventional wisdom. However, the intuitive viewpoint might be restored through some combination of the following mitigating factors:

♦ Splitting the system into small components might reduce the number of lines needed, via reuse. However, the reduction in size would have to be dramatic: in the above example, an 80 percent reduction in size might be necessary. In practice, values considerably less than this have been reported.[11]

♦ The additional unreliability caused by splitting up the system might be due to simple interface inconsistencies. The Basili–Perricone study considered this a possible explanation, as did Moller–Paulish. However, it was not a factor in the Hatton–Hopkins study, since the internally reusable components in the NAG library (largely externally used reusable components) had high interface consistency. Furthermore, it is unlikely to explain the Compton–Withrow data because Ada mandates interface consistency in language implementations. (This may be responsible for the difference in small components in Figure 4.)

♦ The overall maintenance cost might be reduced by modularization, even though the corrective component would cost more. However, some studies[12] report that around 50 percent of all maintenance is corrective, so again this explanation may not be valid. Whatever this contribution might be, this issue has particular relevance for safety-critical systems, where reliability would generally be much more important than ease of change. Thus, even this very simple model has profound implications, as we shall now see.

Start by defining the overall complexity Ω of a software system consisting of N small to medium components, each of implementation complexity Ω_i, by

$$\Omega = \prod_{i=1}^{N} \Omega_i + \int(\Omega_i). \quad (5)$$

Here, the function $f(\Omega_i)$ is an unknown that depends on the combined complexity of the individual components. The product has been used because the logarithmic behavior of the component fault density falls out nicely. In general, for components of small to medium size, we'll assume that the first term on the right-hand side dominates the second term.

Taking the logarithm of Equation 5 gives us

$$E_t = \log \Omega = \log\left\{\prod_{i=1}^{N} \Omega_i \left[1 + \frac{\int(\Omega_i)}{\prod_{i=1}^{N} \Omega_i}\right]\right\} \quad (6)$$

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

$$\left|\frac{\int(\Omega_i)}{\prod_{i=1}^{N} \Omega_i}\right| << 1, \quad (7)$$

Equation 6 becomes

$$E_t = \log \Omega = \log \prod_{i=1}^{N} \Omega_i + \frac{\int(\Omega_i)}{\prod_{i=1}^{N} \Omega_i}. \quad (8)$$

The reason for this rather contrived model is that Equation 8 nicely embodies the logarithmic behavior of Equation 3 as well as the important observation by Basili and Perricone that most of the faults in a real system (89 percent in their case) affect only a single component, and therefore the total number of system faults was approximately the sum of the component faults, neglecting the second term in Equation 8. This model has precisely such behavior, in that the total number of system faults can be approximated by

$$E_t = \log \Omega = \log \prod_{i=1}^{N} \Omega_i = \sum_{i=1}^{N} \log \Omega_i . \quad (9)$$

Based on this model I will make some comparisons with real systems and a few further predictions.

First, assume that a system with E total faults 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, I will further assume that E is an asymptotic value for a stable system in the sense discussed earlier. Then from Equation 9,

$$E = N \log (L/N). \quad (10)$$

Equation 11 yields

$$E_{LOC} \equiv \frac{E}{L} = \frac{N}{L} \log \frac{L}{N}, \quad (11)$$

where E_{LOC} is the number of system faults per line of code. From this, E_{KLOC} (the number of system faults per 1,000 lines of code, as conventionally used) is given simply by

$$E_{KLOC} = 1{,}000\, E_{LOC}. \quad (12)$$

Let's take a closer look at Equations 11 and 12.

Comparing systems with different average component sizes. The first property of Equation 11 to note 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 OO may make things less reliable in terms of total system faults unless the objects are very small indeed. This prediction may, however, be an artifact only and will have to be tested by experiment, because the case histories reported earlier provide no data in this region and thus the approximation used in going from Equation 8 to 9 may be invalid. 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 behavior breaks down and quadratic behavior, observed in the systems studied here, takes hold.

Macroscopic fault behavior may exist in software systems, so there may be limits on the fault density we can achieve.

Total system complexity. The argument that led up to Equation 9 suggests that total system complexity can be modeled approximately as the product of the

original component complexities:

$$\Omega = \prod_{i=1}^{N} \Omega_i. \quad (13)$$

We may be able to predict some software system properties from basic, static parameters like component-size distribution.

Changing an existing system. For an existing system of L lines and N components, differentiating Equation 11 gives

$$\frac{\partial E}{\partial L} = \frac{N}{L} \quad (14)$$

and

$$\frac{\partial E}{\partial L} = \log\left(\frac{L}{N}\right) - 1. \quad (15)$$

This leads to three possibilities for improving the reliability of an existing system, that is, for reducing E:

1. $(\partial E/\partial L) > 0$, keeping N constant and L decreasing. This corresponds simply to reducing the total number of lines for the same number of components. This is essentially the mechanism of reuse.
2. $(\partial E/\partial N) > 0$, keeping L constant and N decreasing. This corresponds to reducing the number of components for the same number of lines. This simply increases the trend to large monolithic components.
3. $(\partial E/\partial N) < 0$, keeping L constant and N increasing. This case holds only for components that are very small already. It suggests that a trend to small components via OO, for example, will improve overall system reliability only if existing components are already very small.

We can estimate the benefit of change 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.

Compelling empirical evidence from disparate sources implies 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 their own results (including myself!).

Given that this behavior spans multiple languages, mature and immature systems, and non–tightly coupled systems (such as the NAG library) and tightly coupled systems (operating systems), this may well be the first quantitative indication that macroscopic fault behavior exists in software systems and thus there may be limits on the fault density we can achieve. It also raises the possibility that we can predict certain software system properties from basic, static parameters like component-size distribution.

The observed qualitative behavior of the systems discussed here was closely predicted by the U-shaped model of component fault density as a function of size. However, further work is needed to quantify complexity in this sense. Using very simple component size distributions, my model predicts the following:

- Some of the case histories cited suggest a maximum fault density (for very small components) as well as a minimum fault density (for rather large components). This offers very tentative evidence that OO may deliver increased reliability, but only if the components are very small, on the order of one or two lines. There is as yet no evidence to support or disprove this conjecture, although defect densities for functional languages may 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 substantial reuse within the same system will likely improve reliability. Modest reuse within a system is likely to make it worse.
- When changing an existing system, the direction of increasing reliability depends on the existing average component size.
- The most reliable systems may be those with component sizes grouped around the 200- to 400-line mark. Bigger and smaller average component sizes appear to degrade reliability.

Further experiments and analysis will be necessary to support or disprove these conjectures. But there is nothing conjectural about the fact that published reliability studies are in serious conflict with the conventional wisdom. In terms of reliability, the structural decomposition of systems into small, easily manageable components does not make a better system, even if they become easier to change, which has not been proven. This has serious implications for high-integrity software development. The apparent existence of a tradeoff between changeability and reliability must be studied further. It may be that this phenomenon pervades other areas of human creativity as well. ◆

ACKNOWLEDGMENTS

I thank my colleagues at Programming Research Ltd. for numerous discussions on this topic, Tom Anderson for stimulating discussions about models for fault behavior, and Norman Fenton for bringing to my attention some of the above case histories, which were instrumental in building the argument. Unknown reviewers made several important suggestions for improvement, as did Tony Hoare who read an early version of these arguments and contributed significantly. Finally, I thank the authors of the published data, without whose efforts no systematic progress can be made.

REFERENCES

1. L. Hatton and T.R. Hopkins, "Experiences with Flint, a Software Metrication Tool for Fortran 77," *Symp. Software Tools*, CSM, Durham, UK, 1989.
2. S. Davey et al., "Metrics Collection in Code and Unit Test as Part of Continuous Quality Improvement," *J. Software Testing, Verification and Reliability*, Vol. 3, 1993, pp. 125-148.
3. K.-H. Moller and D.J. Paulish, "An Empirical Investigation of Software Fault Distribution," *CSR '93*, Chapman-Hall, Amsterdam, 1993.
4. B.T. Compton and C. Withrow, "Prediction and Control of Ada Software Defects," *J. Systems Software*, Vol. 12, 1990, pp. 199-207.
5. V.R. Basili and B.T. Perricone, "Software Errors and Complexity: An Empirical Investigation," *Comm. ACM*, Vol. 1, 1984, pp. 42-52.
6. V.Y. Shen et al., "Identifying Error-Prone Software—An Empirical Study," *IEEE Trans. Software Eng.*, Vol. SE-11, No. 4, 1985, pp. 317-323.
7. B. Kitchenham and P. Mellor, "Data Collection and Analysis," in *Software Metrics: A Rigorous Approach*, N.E. Fenton, ed., Chapman-Hall, London, 1991, pp. 89-110.
8. J.D. Musa, A. Iannino, and K. Okumuto, *Software Reliability*: *Measurement*, *Prediction*, *Application*, McGraw-Hill, New York, 1987.
9. G.A. Miller, "The Magical Number 7 Plus or Minus Two: Some Limits on Our Capacity for Processing Information," *Psychological Rev.*, Vol. 63, 1957, pp. 81-97.
10. E.R. Hilgard, R.C. Atkinson, and R.L. Atkinson, *Introduction to Psychology*, 5th ed., Harcourt Brace Jovanovich, New York, 1971, p. 640.
11. W.B. Frakes and C.J. Fox, "Sixteen Questions on Software Re-use," *Comm. ACM*, Vol. 38, No. 6, 1995, pp. 75-87.
12. R.S. Arnold, *On the Generation and Use of Quantitative Criteria for Assessing Software Maintenance Quality*, Univ. of Maryland microfiche, College Park, Md., 1983.
13. L. Hatton, Safer *C*: *Developing for High-Integrity and Safety-Critical Systems*, McGraw-Hill, New York, 1995.

Les Hatton is a managing partner at Oakwood Computing. Formerly he was director of research for Programming Research Ltd. where the work for this article was done. As a geophysicist he was awarded the European Conrad Schlumberger award in 1987, but now specializes in software safety. He is the author of *Safer C: Software Development in High-Integrity and Safety-Critical Systems* (McGraw-Hill, 1995) and is currently working on a new book entitled *Software Failure: the Avoidable and the Unavoidable*.

Hatton received a BA and MA from King's College, Cambridge, and a PhD from Manchester, all in mathematics.

Address questions about this article to Hatton at Oakwood Computing, Oakwood, 11, Carlton Road, New Malden, Surrey, KT3 3AJ, UK; phone/fax, +44 181-336-1151; e-mail, lesh@oakcomp.cemon.co.uk.