Quantitative Analysis of Faults and Failures in a Complex Software System

Norman E. Fenton, *Member*, *IEEE Computer Society*, and
Niclas Ohlsson, *Member*, *IEEE Computer Society*

Abstract—The dearth of published empirical data on major industrial systems has been one of the reasons that software engineering has failed to establish a proper scientific basis. In this paper, we hope to provide a small contribution to the body of empirical knowledge. We describe a number of results from a quantitative study of faults and failures in two releases of a major commercial system. We tested a range of basic software engineering hypotheses relating to: The Pareto principle of distribution of faults and failures; the use of early fault data to predict later fault and failure data; metrics for fault prediction; and benchmarking fault data. For example, we found strong evidence that a small number of modules contain most of the faults discovered in prerelease testing and that a very small number of modules contain most of the faults discovered in operation. However, in neither case is this explained by the size or complexity of the modules. We found no evidence to support previous claims relating module size to fault density nor did we find evidence that popular complexity metrics are good predictors of either fault-prone or failure-prone modules. We confirmed that the number of faults discovered in prerelease testing is an order of magnitude greater than the number discovered in 12 months of operational use. We also discovered fairly stable numbers of faults discovered at corresponding testing phases. Our most surprising and important result was strong evidence of a counter-intuitive relationship between pre- and postrelease faults: Those modules which are the most fault-prone prerelease are among the least fault-prone postrelease, while conversely, the modules which are most fault-prone postrelease are among the least fault-prone prerelease. This observation has serious ramifications for the commonly used fault density measure. Not only is it misleading to use it as a surrogate quality measure, but, its previous extensive use in metrics studies is shown to be flawed. Our results provide data-points in building up an empirical picture of the software development process. However, even the strong results we have observed are not generally valid as software engineering laws because they fail to take account of basic explanatory data, notably testing effort and operational usage. After all, a module which has not been tested or used will reveal no faults, irrespective of its size, complexity, or any other factor.

Index Terms—Software faults and failures, software metrics, empirical studies.

1 Introduction

Despite some heroic efforts from a small number of research centers and individuals (see, for example, [4], [7], [20], [24], [25], [28], [35], [38]), there continues to be a dearth of published empirical data relating to the quality and reliability of realistic commercial software systems. Two of the most important studies [1] and [3] are now over 16 years old. Adams' study [1] revealed that a great proportion of latent software faults lead to very rare failures in practice, while the vast majority of observed failures are caused by a tiny proportion of the latent faults. Adams observed a remarkably similar distribution of such fault "sizes" across nine different major commercial systems. One conclusion of the Adams' study is that removing large numbers of faults may have a negligible effect on reliability; only when the small proportion of "large" faults are removed will reliability improve significantly. Basili and Pericone [3] looked at a number of factors influencing the fault and failure proneness of modules. One of their most notable results was that larger modules tended to have a lower fault density than smaller ones. Fault density is the number of faults discovered (during some predefined phase of testing or operation) divided by a measure of module size (normally thousands of Lines on Code). While the fault density measure has numerous weaknesses as a quality measure (see [13] for an in-depth discussion of these), this result is nevertheless very surprising. It appears to contradict the very basic hypotheses that underpin the notions of structured and modular programming. Curiously, the same result has been rediscovered in other systems by [30]. Recently, Hatton provided an extensive review of similar empirical studies and came to the conclusion:

> "Compelling empirical evidence from disparate sources implies that in any software system, larger components are proportionally more reliable than smaller components" [17].

Thus, the various empirical studies have thrown up results which are counter-intuitive to the very basic and popular software engineering beliefs. Such studies should have been a warning to the software engineering research community about the importance of establishing a wide empirical basis. Yet, these warnings were clearly not heeded. In [11], we commented on the almost total absence of empirical research on evaluating the effectiveness of different software development and testing methods. There

- *N.E. Fenton is with the RADAR (Risk Assessment and Decision Analysis Research) Group, Computer Science Department, Faculty of Informatics and Mathematical Sciences, Queen Mary and Westfield College, London E1 4NS. E-mail: norman@dcs.qmw.ac.uk.*
- *N. Ohlsson is with the GratisTel International, AB, Gjörwellsgatan 22, 13tr, S-100 26 Stockholm, Sweden. E-mail: niclas.ohlsson@gratistel.com.*

Manuscript received 23 June 1997; accepted 2 Mar. 1999.
Recommended for acceptance by J. Rushby.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number 105280.

also continues to be an almost total absence of published benchmarking data.

In this paper, we hope to provide a very small contribution to the body of empirical knowledge by describing a number of results from a quantitative study of faults and failures in two releases of a major commercial system. We do not claim that the results presented here are novel; on the contrary, we believe that similar analyses are being performed (with similar results) for major systems throughout the world. However, it appears that few organizations are publishing such results, even in the "gray literature" and so there is little if any similar published data. We make no claims about the generalization of these results, but in time we hope they can form part of a broader picture.

In Section 2, we describe the background to the study and the basic data that was collected. In Section 3, we provide pieces of evidence that one day (if a reasonable number of similar studies are published) may help us test some of the most basic of software engineering hypotheses. In particular, we present a range of results and examine the extent to which they provide evidence for or against the following hypotheses.

- Hypotheses relating to the Pareto principle of distribution of faults and failures:
  1. Number of Modules.
     a. A small number of modules contain most of the faults discovered during prerelease testing;
     b. If a small number of modules contain most of the faults discovered during prerelease testing, then this is simply because those modules constitute most of the code size.
  2. Number of Modules.
     a. A small number of modules contain the faults that cause most failures;
     b. If a small number of modules contain most of the operational faults, then this is simply because those modules constitute most of the code size.
- Hypotheses relating to the use of early fault data to predict later fault and failure data (at the *module* level):
  3. A higher incidence of faults in function testing implies a higher incidence of faults in system testing.
  4. A higher incidence of faults in prerelease testing implies higher incidence of failures in operation.

We tested each of these hypotheses from an absolute and normalized fault perspective.

- Hypotheses about metrics for fault prediction:
  5. Simple size metrics, such as Lines of Code (LOC), are good predictors of fault and failure prone modules.
  6. Complexity metrics are better predictors than simple size metrics of fault and failure-prone modules.
- Hypotheses relating to benchmarking figures for quality in terms of defect densities:
  7. Fault densities at corresponding phases of testing and operation remain roughly constant between subsequent major releases of a software system.
  8. Software systems produced in similar environments have broadly similar fault densities at similar testing and operational phases.

For the particular system studied, we provide some evidence for and against some of the above hypotheses and also explain how some previous studies that have looked at these hypotheses are flawed. We stress again, of course, that our study is based on just two releases of a major system and that, therefore, we make no attempt to generalize the results. There is some support for Hypotheses 1a and 2a, while 1b and 2b are rejected. Hypothesis 3 is weakly supported, while, curiously, Hypothesis 4 is strongly rejected. Hypothesis 5 is partly supported, but Hypothesis 6 is weakly rejected for the popular complexity metrics. However, certain complexity metrics which can be extracted from early design specifications are shown to be reasonable fault predictors. Hypothesis 7 is partly supported, while Hypothesis 8 can only be tested properly once other organizations publish analogous results.

We discuss the results in more depth in Section 4. A summary of the results is also provided there (Table 7).

2 The Basic Data

The results reported in this paper are based on empirical data obtained from Ericsson Telecom AB, which develops large software-intensive systems for telecommunication applications. The data presented is based on two major consecutive releases of a large legacy project developing switching systems. Much of the detailed data from the projects are confidential. However, we can report the following background information:

- The development is carried out at more than 20 design centers sited in more than 10 countries.
- A new release typically takes over two years and requires more than 200 man-years of effort.
- The data were collected retrospectively, some manually and others with the support of programs developed by the authors.
- Both releases were approximately the same total system size.

We refer to the earlier of the releases as *release n*, and the later release as *release n+1*. For this study 140 and 246 modules, respectively, from *release n* and *n+1* were selected randomly for analysis from the set of modules that were either new or had been modified. Because of confidentiality, we cannot reveal either the total number of modules or the total size of the respective releases. The size of the random samples were limited by the available effort for

TABLE 1
Distribution of Modules by Size

LOC	Release n	Release n+1
<1000	23	26
1001-2000	58	85
2001-3000	37	73
3001-4000	15	38
4001-5000	6	16
5001-6000	0	6
>6000	1	2
Total	140	246

data-collection and analysis. The reason for the larger sample size in *release n+1* is simply that we were able to automate more of the data collection for this release and, hence, were able to handle more data.

The modules in the samples ranged in size from approximately 1,000 to 6,000 LOC (as shown in Table 1). For *release n* the smallest module was 37 LOC and the largest was 6,580 LOC. For *release n+1* the smallest module was 196 LOC and the largest was 8,458 LOC.

2.1 Dependent Variable

The dependent variable in this study was number of faults. Faults are traced to unique modules. The fault data were collected from four different (nonoverlapping) phases:

- function test (FT);
- system test (ST);
- first 26 weeks at a number of site tests (SI); and
- first year (approximate) of operation after site tests (OP).

Therefore, for each module, we have four corresponding instances of the dependent variable.

The testing process and environment used in this project is well established within the company. It has been developed, maintained, taught, and applied for a number of years. A team separated from the design and implementation organization develops the test cases based on early function specifications.

Throughout the paper, we will refer to the combination of FT and ST faults collectively as *testing faults* or *prerelease faults*. We will refer to the combination of SI and OP faults collectively as *operational faults* or *postrelease faults*.

We shall also refer at times to *failures*. Formally, a failure is an observed deviation of the operational system behavior from specified or expected behavior. All failures are traced back to a unique (operational) fault in a module. Observation of distinct failures that are traced to the same fault are not counted separately. This means, for example, that if 20 OP faults are recorded against module x, then these 20 unique faults caused the set of all failures observed (and which are traced back to faults in module x) during the first year of operation.

The company classified each fault found at any phase according to the following:

1. The fault had already been corrected;
2. The fault will be corrected;
3. The fault requires no action (i.e., not treated as a fault); and
4. The fault was due to installation problems.

In this paper, we have only considered faults classified as 2. Internal investigations have shown that the documentation of faults and their classification according to the above categories is reliable. A summary of the number of faults discovered in each testing phase for each system release is shown in Table 2.

Note that, for all but the Site test phase, there were (as might be expected) approximately twice as many faults recorded in *release n+1* compared to *n*. However, in the Site test phase, there were more than 10 times as many faults. There was no obvious explanation for this wide variation. In any case, in much of the subsequent analysis, we group together the Site and Operation test faults as postrelease faults.

2.2 Independent Variables

Various metrics were collected for each module. These included:

- Lines of code (LOC) as the main size measure;

TABLE 2
Distribution of Faults per Testing Phase

	pre-release faults		post-release faults	
Release	Function test	System test	Site test	Operation
n **(sample size 140 modules)**	916	682	19	52
n+1 **(sample size 246 modules)**	2292	1008	238	108

- McCabe's cyclomatic complexity (CC); and
- Various metrics based on communication (modeled with signals which are similar to messages) between modules and within a module. The most important is the metric *Sig*FF (described in detail in [35]) which is the count of the number of new and modified signals; such signals for each module were specified during the specification phase. In [35], the *Sig*FF metric was also used as a measure of interphase complexity.

The choice of the particular metrics is explained further in Section 3. The complexity metrics were collected automatically from the actual design documents using a tool, ERIMET [34]. This automation was possible as each module was designed using FCTOOL, a tool for the formal description language FDL which is related to SDL's process diagrams [39]. The metrics are extracted directly from the FDL-graphs. The fact that the metrics were computed from artifacts available at the design stage is an important point. It has often been asserted that computing metrics from design documents is far more valuable than metrics from source code [18]. However, there have been very few published attempts to do so. Kitchenham et al. [27], reported on using design metrics, based on Henry and Kafura's information and flow metrics [19], for outlier analysis. Khoshgoftaar et al. [25] used a subset of metrics that "could be collected from design documentation," but the metrics were extracted from the code. Numerous studies, such as [9], [31], have reported using metrics extracted from source code, but few have reported promising prediction results based on design metrics.

2.3 Analysis Techniques

The issue of which statistical tests are relevant for which types of data is discussed in depth in [13]. The key points are that data are measured on different scales (such as *nominal, ordinal, interval,* and *ratio* in increasing order of sophistication) and certain statistical analysis techniques are only meaningful for certain scale types. For example, one of the most common statistics of all, the *mean*, is not a meaningful way to compute the average of a set of data if the data is not at least on an interval scale. In what follows, we have been careful to use only those statistical analysis techniques that are relevant for the scale type of the data. Traditional statistical tests of significance, such as the *t*-test (which assumes an interval or ratio scale for the underlying data) are not appropriate if the data is only ordinal. In our tests of hypothesis below, we use a graphical technique, called Alberg diagrams [35], that assume only that the data is at least ordinal data. Alberg diagrams are used to evaluate the independent variables' ability to rank the dependent variable.

Examples of Alberg diagrams appear in Figs. 3 and 9. Specifically, an Alberg diagram enables us to assess both Type I and Type II errors at the same time for different discriminative thresholds. For example, in Fig. 3, the upper curve shows the percentage of accumulated number of faults when the modules have been sorted in decreasing order with respect to the number of faults (that is, the module with most faults is furthest to the left and the module with fewest faults is furthest to the right). The predictor that should be assessed is used to order the modules in decreasing order with respect to the value of the predictor for each module. Thus, the second curve shows the accumulated percentage of faults for the modules ordered in decreasing order with respect to the predictor value. For a detailed example (and further discussion) of these statistical techniques applied to a similar dataset to the one in this paper, see [36].

3 The Hypotheses Tested and Results

Since the data were collected and analyzed, retrospectively, there was no possibility of setting up any controlled experiments. However, the sheer extent and quality of the data was such that we could use it to test a number of popular software engineering hypotheses relating to the distribution and prediction of faults and failures. In this section, we group the hypotheses into four categories. In Section 3.1, we look at hypotheses relating to the Pareto principle of distribution of faults and failures. It is widely believed, for example, that a small number of modules in any system are likely to contain the majority of the total system faults. This is often referred to as the "20-80 rule" in the sense that 80 percent of the faults are contained in 20 percent of the modules. Despite the widespread belief about this principle, there is little published evidence to support it and the little that there is provides different numbers (ranging from a weak 20-50 to a strong 10-80 type rule). We provide evidence to support the two most commonly cited Pareto principles.

The assumption of the Pareto principle for faults has led many practitioners to seek methods for predicting the fault-prone modules at the earliest possible development and testing phases. These methods seem to fall into two categories:

1. use of early fault data to predict later fault and failure data; and
2. use of product metrics to predict fault and failure data.

Given our evidence to support the Pareto principle, we therefore test a number of hypotheses which relate to these methods of early prediction of fault-prone modules. In Section 3.2, we test hypotheses concerned with 1 above, while, in Section 3.3, we test hypotheses concerned with 2.

Finally, in Section 3.4, we test some hypotheses relating to benchmarking fault data and, at the same time, provide data that can themselves be valuable in future benchmarking studies.

3.1 Hypotheses Relating to the Pareto Principle of Distribution of Faults and Failures

It is widely believed that a relatively small number of all faults or fault types are responsible for the main part of the total cost of poor quality in many different systems. The Pareto principle [44], also called the 20-80 rule, summarizes this notion. The Pareto principle is used to concentrate efforts on the vital few instead of the trivial many. There are a number of examples of the Pareto

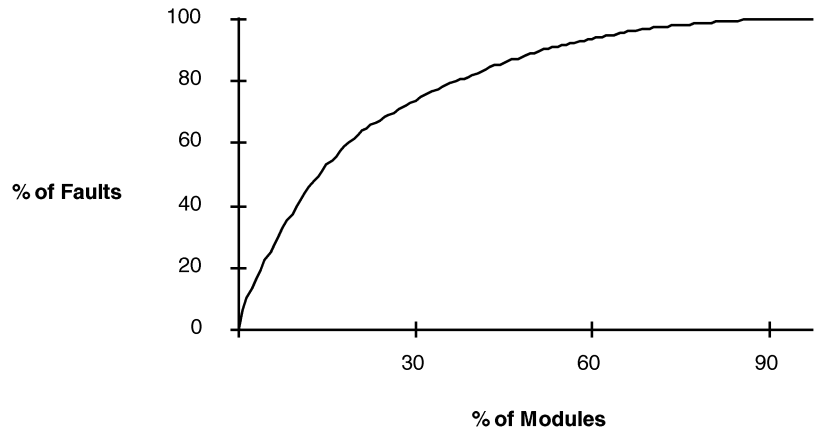

Fig. 1. Pareto diagram showing percentage of modules versus percentage of faults for *release n*.

principle in software engineering. Some of these have gained widespread acceptance, such as the notion that, in any given software system, most faults lie in a small proportion of the software modules. Adams [1] demonstrated that a small number of faults were responsible for a large number of failures. Munson and Khoshgoftaar [31] motivated their discriminative analysis by referring to the 20-80 rule, even though their data demonstrated a 20-65 rule. Zuse [43] used Pareto techniques to identify the most common types of faults found during function testing. Finally, Schulmeyer and McManus [45] described how the principle supports defect identification, inspection, and applied statistical techniques. We investigated four related Pareto hypotheses:

Hypothesis 1a. A small number of modules contain most of the faults discovered during prerelease testing (phases FT and ST).

Hypothesis 1b. If a small number of modules contain most of the faults discovered during prerelease testing, then this is simply because those modules constitute most of the code size.

Hypothesis 2a. A small number of modules contain most of the operational faults (meaning failures as we have defined them above observed in phases SI and OP).

Hypothesis 2b. If a small number of modules contain most of the operational faults, then this is simply because those modules constitute most of the code size.

We now examine each of these in turn.

3.1.1 Hypothesis 1a

Fig. 1 illustrates that 20 percent of the modules were responsible for nearly 60 percent of the faults found in testing for *release n*. An almost identical result was obtained for *release n+1*, but is not shown here. This is also almost identical to the result in earlier work where the faults from both testing and operation were considered [35]. This, together with results such as [31], [20], provides support for Hypothesis 1a and even suggests a specific Pareto distribution in the area of 20-60. It is worth noting that this 20-60 finding is not as strong as the one observed by [6] (they found that 12 percent of the modules, referred to as packages, accounted for 75 percent of all the faults during system integration and test).

3.1.2 Hypothesis 1b

Since we found strong support for Hypothesis 1a, it makes sense to test Hypothesis 1b. It is popularly believed that Hypothesis 1a is easily explained away by the fact that the small proportion of modules causing all the faults actually constitute most of the system size. For example, [6] found that the 12 percent of modules accounting for 75 percent of the faults accounted for 63 percent of the LOC. In our study, we found no evidence to support this Hypotheses 1b. For *release n*, the 20 percent of the modules which account for 60 percent of the faults (discussed in Hypothesis 1a) actually make up just 30 percent of the system size. The result for *release n+1* was almost identical. This compares with the result in [20] where the 38 percent of modules that account for 80 percent of the faults actually constitute 54 percent of the code.

3.1.3 Hypothesis 2a

We discovered not just support for a Pareto distribution, but a much more exaggerated one than for Hypothesis 1a. Fig. 2 illustrates this Pareto effect in *release n*. Here 10 percent of the modules were responsible for 100 percent of the failures found. The result for *release n+1* is not so remarkable, but is nevertheless still quite striking: 10 percent of the modules were responsible for 80 percent of the failures.

3.1.4 Hypothesis 2b

As with Hypothesis 1a, it is popularly believed that Hypothesis 2a is easily explained away by the fact that the small proportion of modules causing all the failures actually constitute most of the system size. In fact, not only did we find no evidence for Hypothesis 2b, but we discovered strong evidence in favor of a converse hypothesis: *Most operational faults are caused by faults in a small proportion of the code.*

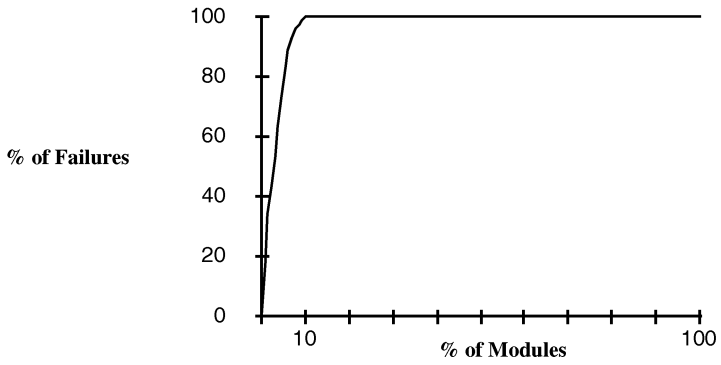

Fig. 2. Pareto diagram showing percentage of modules versus percentage of failures for *release n*.

For *release n*, 100 percent of the operational faults are contained in modules that make up just 12 percent of the entire system size. For *release n+1*, 60 percent of the operational faults were contained in modules that make up just 6 percent of the entire system size, while 78 percent of the operational faults were contained in modules that make up 10 percent of the entire system size.

3.2 Hypotheses Relating to the Use of Early Fault Data to Predict Later Fault and Failure Data

Given the likelihood of Hypotheses 1a and 2a, there is a strong case for trying to predict the most fault-prone modules as early as possible during development. In this and the next subsection, we test hypotheses relating to methods of doing precisely that. First, we look at the use of fault data collected early as a means of predicting subsequent faults and failures. Specifically, we test the hypotheses:

Hypothesis 3. Higher incidence of faults in function testing (FT) implies higher incidence of faults in system testing (ST)

Hypothesis 4. Higher incidence of faults in all prerelease testing (FT and ST) implies higher incidence of faults in postrelease operation (SI and OP).

We tested each of these hypotheses from an absolute and normalized fault perspective. We now examine the results.

3.2.1 Hypothesis 3

The results associated with this hypothesis are not very strong. In *release n* (see the Alberg diagram in Fig. 3), 50 percent of the faults in system test occurred in modules which were responsible for 37 percent of the faults in function test.

From a prediction perspective, the figures indicate that the most fault-prone modules during function test will, to some extent, also be fault-prone in system test. However, 10 percent of the most fault-prone modules in system test are responsible for 38 percent of the faults in system test, but 10 percent of the most fault-prone modules in function test are only responsible for 17 percent of the faults in system test. This is persistent up to 75 percent of the modules. This means that nearly 20 percent of the faults in system test need to be explained in another way. The same pattern was found when using normalized data (faults/LOC) instead of absolute, even though the percentages were generally lower and the predictions a bit poorer. In Fig. 4, the data are presented as a simple scatterplot.

The results were only slightly different for *release n+1*, where we found:

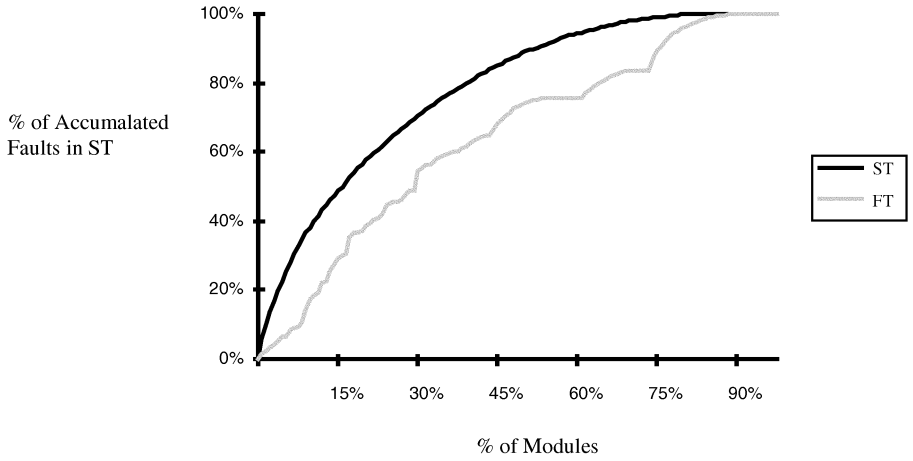

Fig. 3. Accumulated percentage of the absolute number faults in system test when modules are ordered with respect to the number of faults in system test and function test for *release n*.

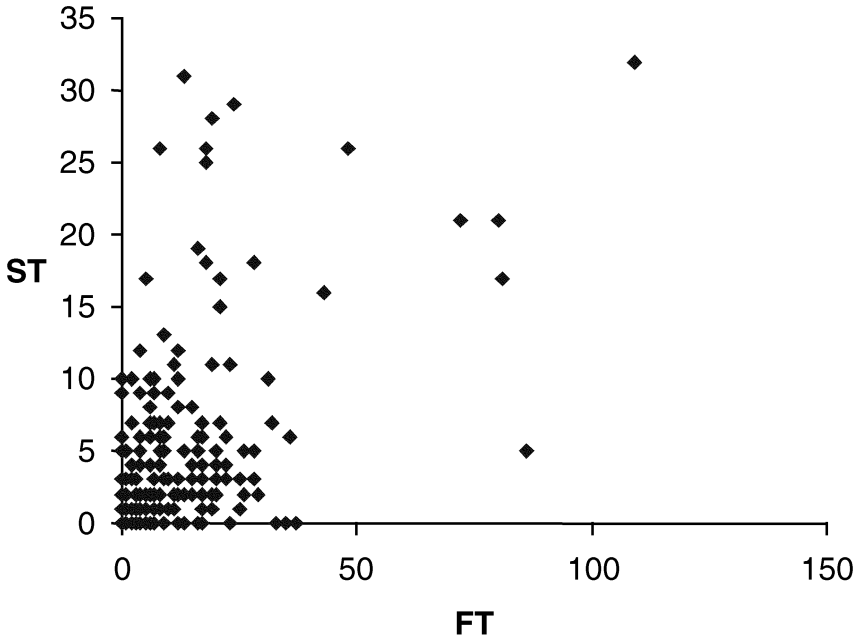

Fig. 4. Scatter plot of faults found in Function test (FT) against faults found in System Test (ST) for *release n*—each dot represents a module.

- Fifty percent of the faults in system test occurred in modules which were responsible for 25 percent of the faults in function test.
- Ten percent of the most fault-prone modules in system test are responsible for 46 percent of the faults in system test, but 10 percent of the most fault-prone modules in function test are only responsible for 24 percent of the faults in system test.

These results (and also when using normalized data instead of absolute) are very similar to the result in *release n*.

3.2.2 Hypothesis 4

The rationale behind Hypothesis 4 is the belief in the inevitability of "rogue modules," a relatively small proportion of modules in a system that account for most of the faults and which are likely to be fault-prone both pre- and postrelease. It is often assumed that such modules are somehow intrinsically complex or generally poorly built. "If you want to find where the faults lie, look where you found them in the past" is a very common and popular maxim. For example, Compton and Withrow [6] have found as much as six times greater post delivery defect density when analyzing modules with faults discovered prior to delivery.

The hypothesis is by no means *universally* accepted. There are some who acknowledge that a high incidence of faults in a module prior to release may simply confirm that such a module has been well tested and will, therefore, be reliable in operation. However, the literature on metrics validation [12] confirms fairly widespread acceptance of the hypothesis despite little empirical evidence to support it. Specifically, many metrics validation studies have used prerelease module *fault density* (the number of faults found in the module divided by the size of the module) as a surrogate measure for how reliable the module is in operation.

In many respects, the results in our study relating to this hypothesis are the most remarkable of all. Not only is there no evidence to support the hypothesis, but there is evidence to support a converse hypothesis. In both *release n* and *n+1* almost all of the faults discovered in prerelease testing appear in modules which subsequently reveal almost no operation faults. Specifically, we found:

- In *release n* (see Fig. 5), 93 percent of faults in prerelease testing occur in modules which have NO subsequent operational faults (of which there were 75 in total). Thus, 100 percent of the 75 failures in operation occur in modules which account for just 7 percent of the faults discovered in prerelease testing.
- In *release n+1*, we observed a much greater number of operational faults, but a similar phenomenon to that of *release n* (see Fig. 6). Some 77 percent of prerelease faults occur in modules which have NO postrelease faults. Thus, 100 percent of the 366 failures in operation occur in modules which account for just 23 percent of the faults discovered in function and system test.

These remarkable results are also closely related to the Adams' phenomenon. The results have major ramifications for the commonly used software measure, *fault density*. Specifically, it appears that modules with high fault density prerelease are likely to have low fault-density postrelease and vice versa. We discuss the implications at length in Section 4, including the ramifications on previous metrics validation studies.

3.3 Hypotheses about Metrics for Fault Prediction

In the previous subsection, we were concerned with using early fault counts to predict subsequent fault prone modules. In the absence of early fault data, it has been

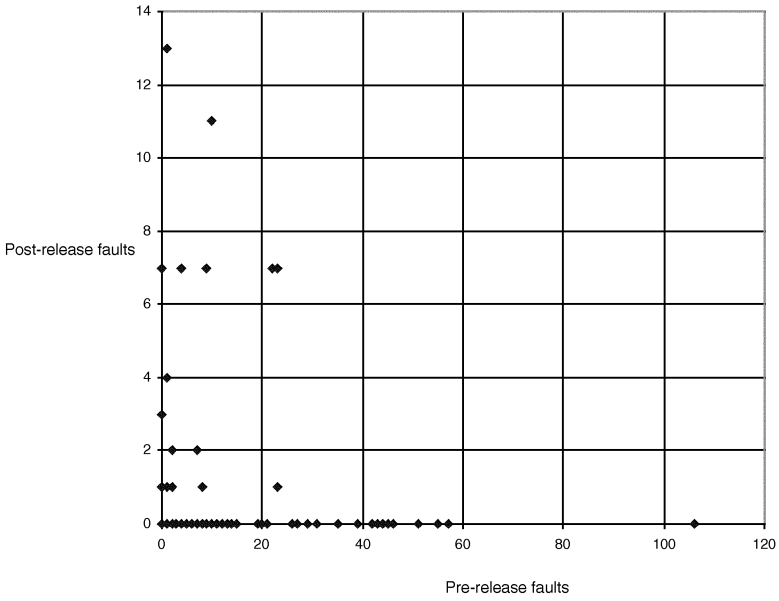

Fig. 5. Scatter plot of prerelease faults against postrelease faults for *release n* (each dot represents a module).

widely proposed that software metrics (which can be automatically computed from module designs or code) can be used to predict fault prone modules. In fact, this is widely considered to be the major benefit of such metrics [13]. We therefore attempted to test the basic hypotheses which underpin these assumptions. Specifically, we tested:

Hypothesis 5. Size metrics (such as LOC) are good predictors of fault- and failure-prone modules.

Hypothesis 6. Complexity metrics are better predictors than simple size metrics, especially at predicting fault-prone modules.

3.3.1 Hypothesis 5

Strictly speaking, we have to test several different, but closely related, hypotheses:

Hypothesis 5a. Smaller modules are less likely to be failure-prone than larger ones.

Hypothesis 5b. Size metrics (such as LOC) are good predictors of number of prerelease faults in a module.

Hypothesis 5c. Size metrics (such as LOC) are good predictors of number of postrelease faults in a module.

Hypothesis 5d. Size metrics (such as LOC) are good predictors of a module's (prerelease) fault-density.

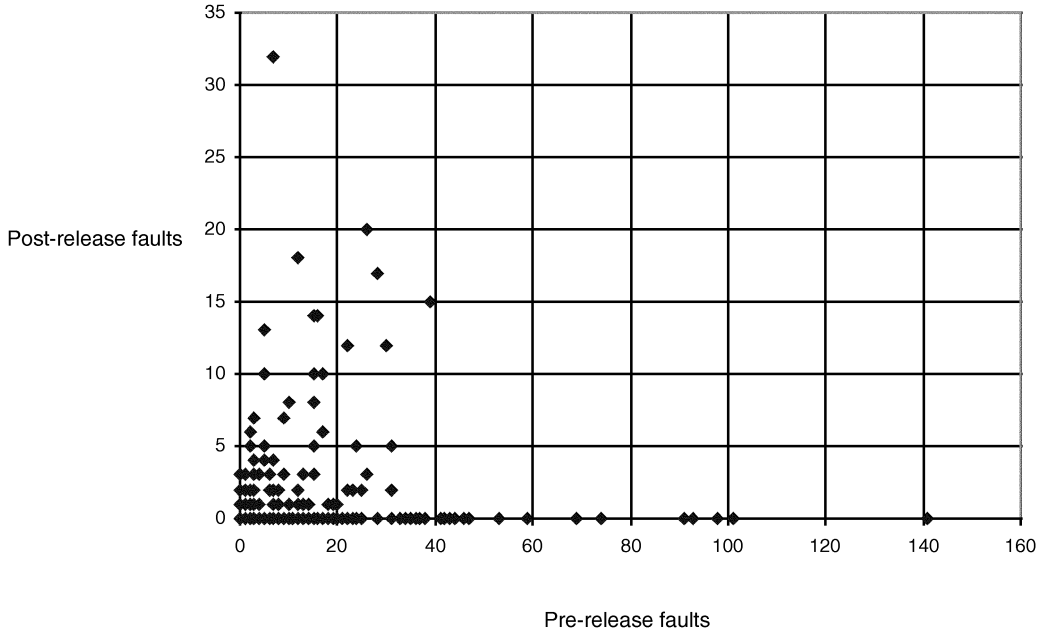

Fig. 6. Scatter plot of prerelease faults against postrelease faults for *release n+1* (each dot represents a module).

TABLE 3
Number of Modules Affected by a Fault for *Release n* (140 Modules, 1,815 Faults) and *Release n+1* (246 Modules, 3,795 Faults)

	Release n			Release n+1			
Fault	Modified	New	Percent modified modules	Modified	New	Split	Percent modified modules
0	9	0	7	15	3	0	7
1	5	3	4	16	1	0	7
2	12	0	9	18	2	0	8
3	10	0	8	13	0	0	6
4	8	0	6	12	1	0	5
5	12	1	9	7	0	0	3
6	3	1	2	14	1	0	6
7	4	0	3	5	0	0	2
8	7	0	5	5	0	1	2
9	8	2	6	13	0	1	6
10	5	0	4	6	2	0	3
11 to 15	17	1	13	24	1	0	11
16 to 20	4	0	3	14	2	3	6
21 to 25	3	0	2	21	0	0	9
26 to 30	7	0	5	9	0	1	4
31 to 35	5	0	4	8	0	1	4
36 to 40	2	0	2	6	0	0	3
>40	9	2	7	18	0	2	8

Hypothesis 5e. Size metrics (such as LOC) are good predictors of a module's (postrelease) fault-density.

Hypothesis 5a underpins, in many respects, the principles behind most modern programming methods, such as modular, structured, and object-oriented. The general idea has been that smaller modules should be easier to develop, test, and maintain, thereby leading to fewer operational faults in them. On the other hand, it is also accepted that if modules are made too small, then all the complexity is pushed into the interface/communication mechanisms. Size guidelines for decomposing a system into modules are therefore desirable for most organizations.

It turns out that the small number of relevant empirical studies have produced counter-intuitive results about the relationship between size and (operational) fault density. Basili and Pericone [3] reported that fault density appeared to *decrease* with module size. Their explanation for this was the large number of interface faults were spread equally across all modules. The relatively high proportion of small modules were also offered as an explanation. Other authors, such as Moller and Paulish [30] who observed a similar trend, suggested that larger modules tended to be under better configuration management than smaller ones which tended to be produced "on the fly." In fact, our study did not reveal any similar trend and we believe the strong results of the previous studies may be due to inappropriate analyses.

We tested Hypothesis 5a by replicating the key part of the Basili and Perricone study [3]. Table 3 (which compares with Basili and Perricone's Table III) shows the number of modules that had a certain number of faults. The table also displays the figures for the different types of modules and the percentages. The table groups modules according to the frequency of faults found. Thus, the first row considers those modules for which zero faults were found, the second row considers those modules for which one fault was found, etc. For example, reading the first row (about modules that had zero faults), we find that:

- For *release n* there were nine modified modules (with zero faults) and zero new modules (with zero faults). The nine modified modules made up 7 percent of all the modified modules.
- For *release n+1* there were 15 modified modules (with zero faults) and three new modules (with zero faults). The 15 modified modules made up 7 percent of all the modified modules. Additionally, for this release a number of modules that existed before the project were split into two modules. The data for these are listed in the column labelled "Split."

The data set analyzed in this paper has, in comparison with [3], a lower proportion of modules with few faults and the proportion of new modules is lower. In subsequent analysis, all new modules have been excluded. The modules are also generally larger than those in [3], but we do not believe this introduces any bias.

Hypotheses 5b and c are tested with the two respective scatter plots for lines of code versus the number of pre- and postrelease faults in Fig. 7. As the figures show, there is no strong evidence of trends for *release n+1*. Neither could any strong trends be observed when line of code versus the total

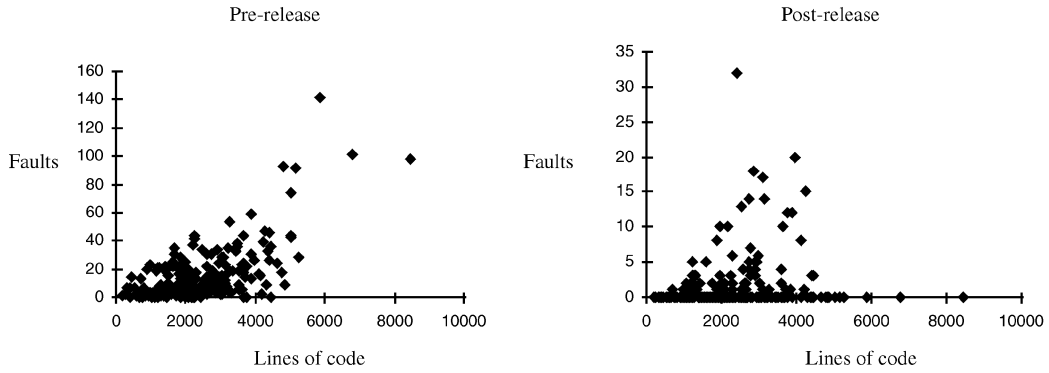

Fig. 7. Scatterplots of LOC against pre- and postrelease faults for *release n+1* (each dot represents a module).

number of faults were graphed in Fig. 8. The results for *release n* were reasonably similar.

When Basili and Pericone could not see any trend, they calculated the number of faults per 1,000 executable lines of code. Table 4 (which compares with Table VII in [3]) shows these results for our study.

Superficially, the results in Table 4 for *release n+1* appear to support the Basili and Pericone finding. In *release n+1* it is clear that the smallest modules have the highest fault density. However, the fault density is very similar for the other groups. For *release n*, the result is the opposite of what was reported by Basili and Perricone. The approach to grouping data as done in [3] is highly misleading. Basili and Pericone did not show a simple plot of fault density against module size, as we have done in Fig. 9 for *release n+1*. Even though the grouped data for this release appeared to support the Basili and Pericone findings, this graph shows only a very high variation for the small modules and no evidence that module size has a significant impact on fault-density. Nor could we find support for Hypotheses 5e and 5d. Clearly, other explanatory factors, such as design, inspection, and testing effort per module, will be more important. This is discussed in depth in [46].

The LOC ranking ability is assessed in the Albert diagram of Fig. 10. The diagram reveals that, even though previous analysis did not indicate any predictability, LOC is quite good at ranking the most fault-prone modules and, for the most fault prone-modules (the 20 percent), much better, than any previous ones.

3.3.2 Hypothesis 6

"Complexity metrics" is the rather misleading term used to describe a class of measures that can be extracted directly from source code (or some structural model of it, like a flowgraph representation). Occasionally (and more beneficially), complexity metrics can be extracted before code is produced, such as when the detailed designs are represented in a graphical language like SDL (as was the case for the system in this study). The archetypal complexity metric is McCabe's cyclomatic number [29], but there have in fact been many dozens that have been published [43]. The details and, also, the limitations of complexity metrics have been extensively documented (see [13]) and we do not wish to revisit those issues here. What we are concerned with here is the underlying assumption that the most commonly used complexity metrics are useful because they are (easy to extract) indicators of where the faults lie in a system. For example, Munson and Khosghoftaar asserted:

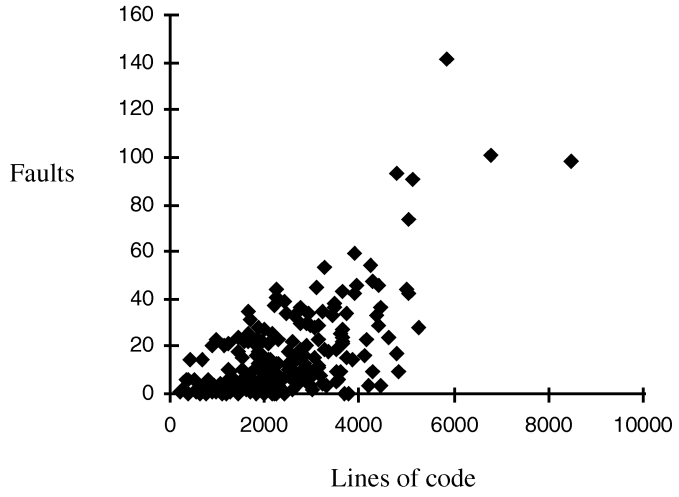

Fig. 8. Scatterplots of LOC against all faults for *release n+1* (each dot represents a module).

TABLE 4
Faults/1,000 Lines of Code *Release n* and *n+1*

	Release n		Release n+1	
Module size	Frequency	Faults/1000 Lines	Frequency	Faults/1000 Lines
500	3	1.45	6	13
1000	15	4.77	17	6
1500	32	5.24	35	5
2000	24	6.32	41	7
2500	14	5.88	34	5
3000	22	5.74	37	5
3500	11	7.83	18	7
>3500	9	7.38	42	8

> "There is a clear intuitive basis for believing that complex programs have more faults in them than simple programs" [31].

An implicit assumption is that complexity metrics are better than simple size measures in this respect (for, if not, there is little motivation to use them). There have been many investigations into the ability of different complexity metrics to predict fault-proneness. Most studies have concentrated on McCabe's cyclomatic complexity and the Halstead metrics. The results of these studies are mixed. There is still no conclusive evidence that such crude metrics are better predictors of fault-proneness than something as simple as LOC. It was because of this that researchers have long been investigating more discriminating and sensible complexity metrics. Nevertheless, cyclomatic complexity remains exceptionally popular. It is easily computed by static analysis (unlike most of the more discriminating metrics) and it is widely used for quality control purposes. For example, Grady [15] reports that, at Hewlett Packard, any modules with a cyclomatic complexity higher than 16 must be redesigned. Cyclomatic complexity, along with some more relevant complexity metrics (based on *Sig*FF), were routinely collected in our case study system. Because of the continued widespread popularity of the cyclomatic complexity metric, together with the doubts about its true validity, we now investigate its validity with respect to it fault prediction power. Thus, the results here provide another validation study (but one which we argue is based on more realistic fault data). We also perform a similar analysis of the *Sig*FF-based metrics described in Section 2.2.

In testing Hypothesis 5, we demonstrated the problem with comparing average figures for different size intervals. Instead of replicating the relevant analysis in [3] by calculating the average cyclomatic number for each module size class and then plotting the results, we just generated scatter plots and Alberg diagrams.

When the cyclomatic complexity and the pre- and postrelease faults were graphed for *release n+1* (Fig. 11) we observed a number of interesting trends. The most complex modules appear to be more fault-prone in prerelease, but appear to have nearly no faults in post-release. The most fault-prone modules in postrelease appear to be the less complex modules. This could be explained by how test effort is distributed over the modules: Modules that appear to be complex are treated with extra care than simpler ones. Analyzing retrospectively, the earlier graphs for size versus faults reveal a similar pattern.

The scatter plot for the cyclomatic complexity and the total number of faults (Fig. 12) shows again some small

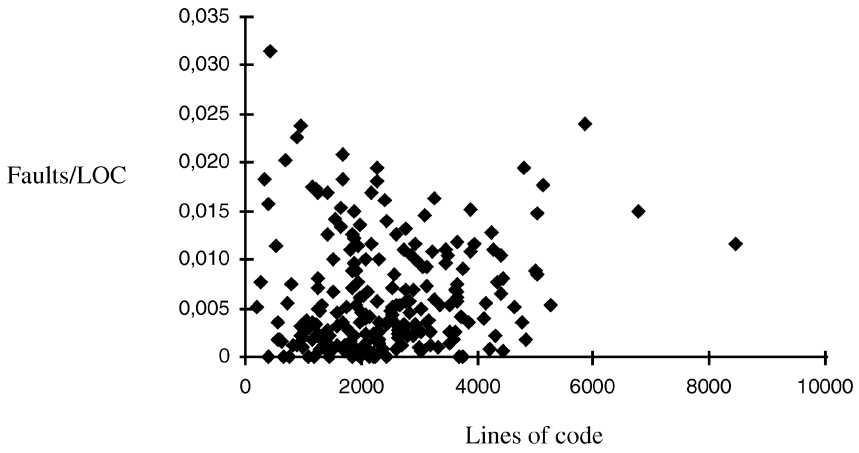

Fig. 9. Scatterplot of module fault density against size for *release n+1*.

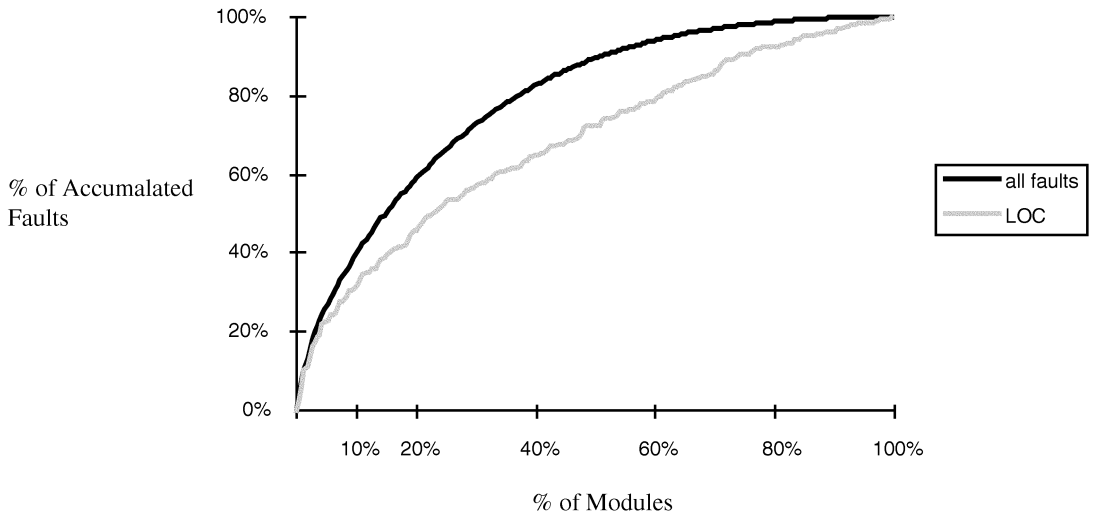

Fig. 10. Accumulated percentage of the absolute number of all faults when modules are ordered with respect to LOC for *release n+1*.

indication of correlation. The Alberg diagrams were similar as when size was used.

To explore the relations further the scatter plots were also graphed with normalized data (Fig. 13). The result showed even more clearly that the most fault-prone modules in prerelease have nearly no postrelease faults.

In order to determine whether or not large modules were less dense or complex than smaller modules, [3] plotted the cyclomatic complexity versus module size. Following this same pattern, in earlier analysis, they failed to see any trends and, therefore, they analyzed the relation by grouping modules according to size. As illustrated in Fig. 13, this can be very misleading. Instead, we graphed scatter plots of the relation and calculated the correlation (Fig. 14).

The relation may not be linear. However, there is a good linear correlation between cyclomatic complexity and LOC^2 (specifically it is 0.79).

Earlier studies [35] have suggested that other design metrics could be used in combination or on their own to explain fault-proneness. Therefore, we did the same analysis using the *Sig*FF measure instead of cyclomatic complexity.

The scatterplots using absolute numbers (Fig. 15), or normalized data did not indicate any new trends. In earlier work, the product of cyclomatic complexity (CC) and *Sig*FF was shown to be a good predictor of fault-proneness. To evaluate CC**Sig*FF predictability, the Alberg diagram was graphed (Fig. 16). The combined metrics appear to be better than both *Sig*FF and Cyclomatic Complexity on their own, and also better than the size metric.

The above results do not paint a very glowing report of the usefulness of complexity metrics. However, it can be argued that "being a good predictor of fault density" (even postrelease fault density) is not an especially appropriate validation criteria for complexity metrics. Since "complexity" infers a notion of difficulty of understanding, the most natural validation criteria for such metrics should be that they are good predictors of maintainability (as measured, for example, by the time it takes to repair and fix faults in the given module). This is discussed in [11]. Nevertheless, there are some positive aspects. The combined metric CC**Sig*FF is again shown to be a reasonable predictor of fault-prone modules. Also, measures like *Sig*FF are, unlike LOC, available at a very early stage in the software development. The fact that it correlates so closely with the final LOC *and* is a good predictor of total number of faults is a major benefit.

3.4 Hypotheses Relating to Benchmarking

One of the major benefits of collecting and publicizing the kind of data discussed in this paper is to enable both intra- and intercompany comparisons. Despite the incredibly vast

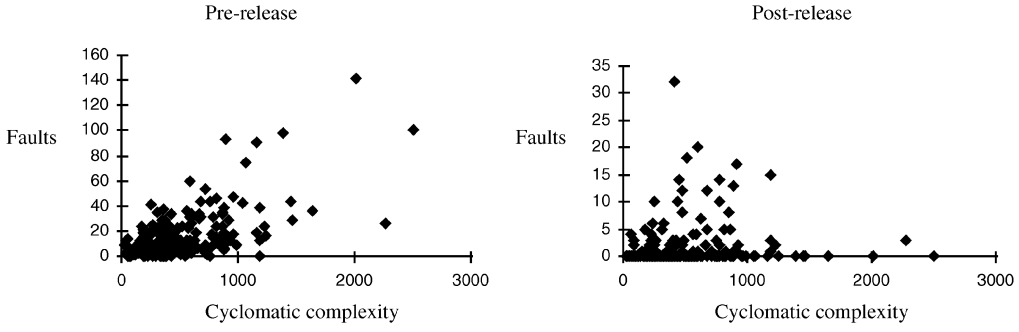

Fig. 11. Scatterplots of cyclomatic complexity against number of pre- and postrelease faults for *release n+1* (each dot represents a module).

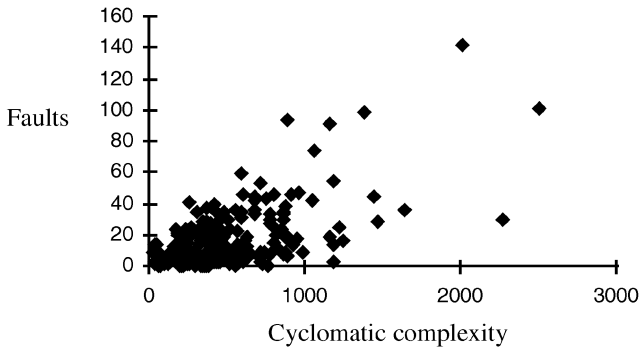

Fig. 12. Scatterplot of cyclomatic complexity against all faults for *release n+1* (each dot represents a module).

volumes of software in operation throughout the world, there is no consensus about what constitutes, for example, a good, bad, or average fault density under certain fixed conditions of measurement. It does not seem unreasonable to assume that such information might be known, for example, for commercial C programs where faults are defined as operational faults (in the sense of this paper) during the first 12 months of use by a typical user. Although individual companies may know this kind of data for their own systems, little has been published. The "gray" literature (as referenced, for example, in [8], [14], [37]) seems to suggest some crude (but not fully substantiated) guidelines, such as the following, for fault density in the first 12 months of typical operational use:

- Less that one fault per thousand lines of code (KLOC) is very good—and typically only achieved by companies using state-of-the-art development and testing methods;
- Between three to eight faults per KLOC is typical; and
- Greater than 10 faults per KLOC is bad.

In one of the few fully documented studies, Kaaniche and Kanour [20] report on a fault density of just under three faults per KLOC for a major telecommunications system, but this is measured over five years of operational use.

When prerelease faults only are considered, there is some notion that 10-30 faults per KLOC is typical for function, system, and integration testing combined. For reasons discussed already, high values of prerelease fault density is not indicative of poor quality (and may, in fact, suggest the opposite). Therefore, it would be churlish to talk in terms of "good" and "bad" fault densities because, as we have already stressed, these figures may be explained by key factors such as the effort spent on testing.

In this study, since we have data on successive releases, we can consider the following hypothesis:

Hypothesis 7. Fault densities at corresponding phases of testing and operation remain roughly constant between subsequent major releases of a software system.

The results we present, being based only on one system, represent just a single data-point, but, nevertheless, we

post-release faults.

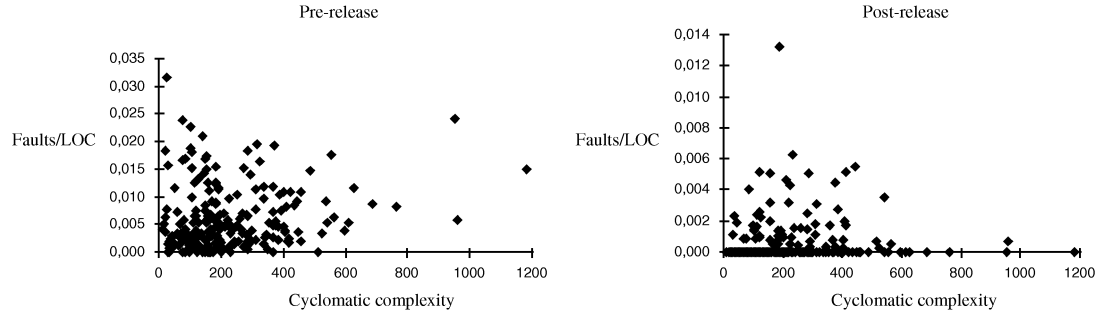

Fig. 13. Scatterplots of cyclomatic complexity against fault density (pre- and postrelease) for *release n+1* (each dot represents a module).

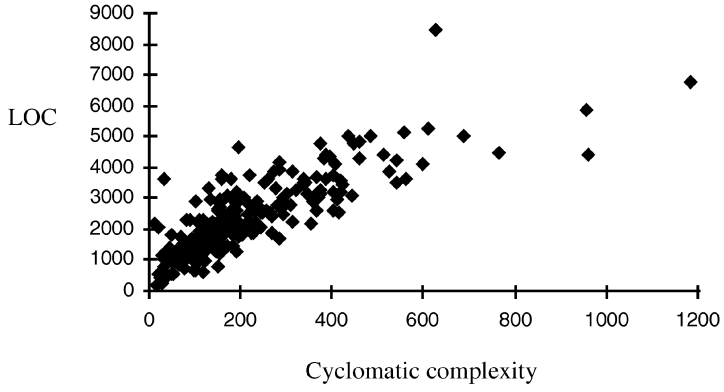

Fig. 14. Complexity versus module size.

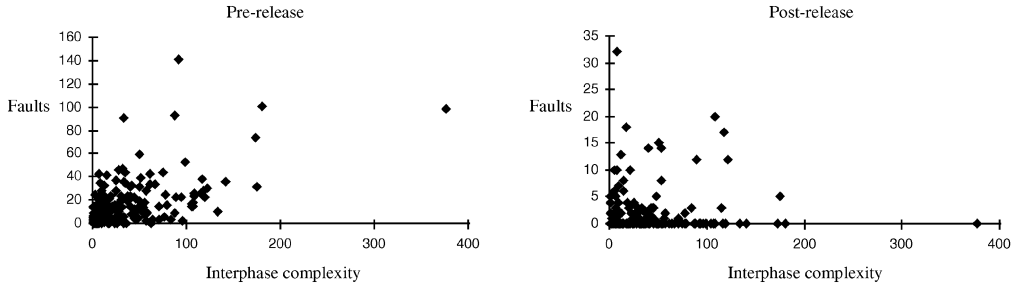

Fig. 15. Scatterplots of *Sig*FF against number of pre- and postrelease faults for *release n+1* (each dot represents a module).

believe it may also be valuable for other researchers. In a similar vein, we consider:

Hypothesis 8. Software systems produced in similar environments have broadly similar fault densities at similar testing and operational phases.

Really, we are hoping to build up an idea of the *range* of fault densities that can reasonably be expected. We compare our results with some other published data.

3.4.1 Hypothesis 8

As Table 5 shows, there is some support for the hypothesis that the fault-density remains roughly the same between subsequent releases. The only exceptional phase is SI. As well as providing some support for the hypothesis, the result suggests that the development process is stable and repeatable with respect to the fault-density.

3.4.2 Hypothesis 9

To test this hypothesis, we compared the results of this case study with other published data. For simplicity, we restricted our analysis to the two distinct phases: 1) prerelease fault density; and 2) postrelease fault density. First, we can compare the two results of the two separate releases in the cases study (Table 6).

The overall fault densities are similar to those reported for a range of systems in [16], while [2] reported similar ball-park figures in a study of Ada programs, 3.0 to 5.5 faults/KLOC. The postrelease fault densities seem to be roughly in line of those reported studies of best practice.

More interesting is the difference between the pre- and postrelease fault densities. In both versions, the prerelease fault density is an order of magnitude higher than the postrelease fault density.

Of the few published studies that reveal the difference between pre- and postrelease fault density, Pfleeger and Hatton [37] also report 10 times as many faults in prerelease (although the overall fault density is lower). Kitchenham et al. [26] reports a higher ratio of prerelease to postrelease. Their study was an investigation into the impact of inspections; combining the inspected and noninspected code together reveals a prerelease fault density of approximately 16 per KLOC and a postrelease fault density of approximately 0.3 per KLOC. However, it is likely that the operational time here was not as long. Kaaniche and Kanoun [20] report a very different picture of pre- and postrelease fault densities; in their paper, the average fault density over all modules appears to be slightly higher postrelease than prerelease. However, this can be explained by the fact that "prerelease" testing in this case is restricted to something analogous to our system test, while the postrelease period is some five years of operational use.

4 Discussion and Conclusions

Apart from the usual quality control angle, a very important perceived benefit of collecting fault data at different testing phases is to be able to move toward statistical process

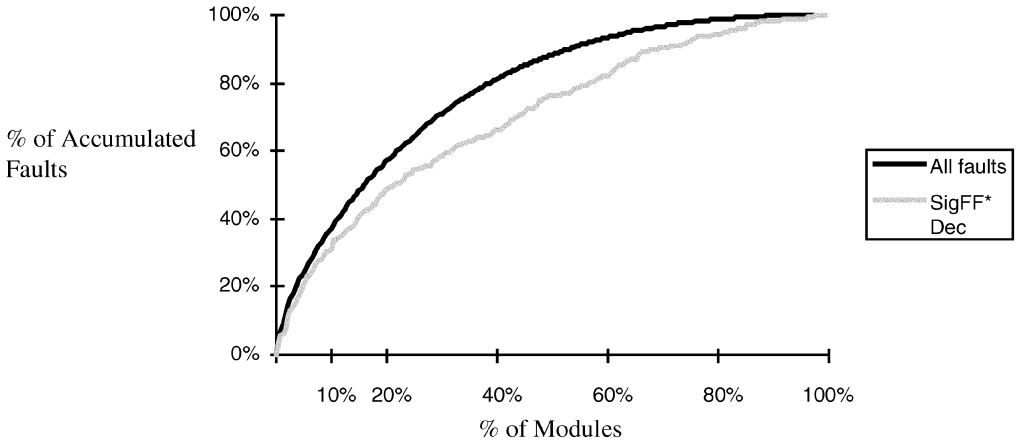

Fig. 16. Accumulated percentage of the absolute number of all faults when modules are ordered with respect to LOC for *release n+1*.

TABLE 5
Fault Densities at the Four Phases of Testing and Operation

	FT	ST	SI	OP
Rel n	3.49	2.60	0.07	0.20
Rel n+1	4.15	1.82	0.43	0.20

TABLE 6
Fault Densities Pre- and Postrelease for the Case Study System

	Pre-release	Post-release	All
Rel n	6.09	0.27	6.36
Rel n+1	5.97	0.63	6.60

control for software development. For example, this is the basis for the software factory approach proposed by Japanese companies, such as Hitachi [41], in which they build fault profiles that enable them to claim accurate fault and failure prediction. Another important motivation for collecting the various fault data is to enable us to evaluate the effectiveness of different testing strategies. In this paper, we have used an extensive example of fault and failure data to test a range of popular software engineering hypotheses.

The results we have presented (which are summarized in Table 7) come from just two releases of a major system developed by a single organization. We make no claims about the generalization of these results. However, given the rigor and extensiveness of the data-collection and also the strength of some of the observations, we feel that there are lessons to be learned by the wider community.

The evidence we found in support of the two Pareto principles 1a and 2a is the least surprising, but there was previously little published empirical data to support it. It does seem to be inevitable that a small number of the modules in a system will contain a large proportion of the prerelease faults and that a small proportion of the modules will contain a large proportion of the postrelease faults. However, the popularly believed explanations for these two phenomena were not supported in this case:

- It is *not* the case that size explains in any significant way the number of faults. Many people seem to believe (Hypotheses 1b and 2b) that the reason why a small proportion of modules account for most faults is simply because those fault-prone modules are disproportionately large and therefore account for most of the system size. We have shown this assumption to be false for this system.
- Nor is it the case that "complexity" (or at least complexity as measured by "complexity metrics") explains the fault-prone behavior (Hypothesis 6). In fact complexity is not significantly better at predicting fault- and failure-prone modules than simple size measures.
- It is also *not* the case that the set of modules which are especially fault-prone prerelease are going to be roughly the same set of modules that are especially fault-prone postrelease (Hypothesis 4). Yet this view seems to be widely accepted, partly on the assumption that certain modules are "intrinsically" difficult and will be so throughout their testing and operational life.

Our strong rejection of Hypothesis 4 in this case has some important ramifications. Many believe that the first place to look for modules likely to be fault-prone in operation is in those modules which were fault-prone during testing. In fact, our results relating to Hypothesis 4 suggest exactly the opposite testing strategy may be the most effective. If you want to find the modules likely to be fault-prone in operation, then you should ignore all the modules which were fault-prone in testing! In reality, the danger here is in assuming that the given data provides evidence of a *causal* relationship. The data we observed can be explained by the fact that the modules in which few faults are discovered during testing may simply not have been tested properly. Those modules which reveal large numbers of faults during testing may genuinely be very well tested in the sense that *all* the faults really are "tested out of them." The key missing explanatory data in this case is, of course, *testing effort*, which was unfortunately not available to us in this case study.

The results of Hypothesis 4 also bring into question the entire rationale for the way software complexity metrics are used and validated. The ultimate aim of complexity metrics is to predict modules which are fault-prone *postrelease*. Yet we have found that there is no relationship between the modules which are fault-prone prerelease and the modules which are fault-prone postrelease. Most previous "validation" studies of complexity metrics have deemed a metric "valid" if it correlates with the (prerelease) fault density. Our results suggest that "valid" metrics may therefore be inherently poor at predicting what they are supposed to predict. The results of Hypothesis 4 also highlight the dangers of using fault density as a de facto measure of user perceived *software quality*. If fault density is measured in terms of prerelease faults (as is very common), then, at the module level, this measure tells us worse than nothing about the quality of the module; a high value is more likely to be an indicator of extensive testing than of poor quality.

Our analysis of the value of "complexity" metrics is mixed. We confirmed some previous studies' results that popular complexity metrics are closely correlated to size metrics like LOC. While LOC (and, hence, also the complexity metrics) are reasonable predictors of absolute number of faults, they are very poor predictors of fault density (which is what we are really after). However, some complexity metrics, like *Sig*FF, are, unlike LOC available at a very early stage in the software development process. The fact that it correlates so closely with the final LOC, is therefore very useful. Moreover, we argued [13] that being a good predictor of fault-proneness may not be the most appropriate test of "validity" of a complexity metric. It is more reasonable to expect complexity metrics to be good predictors of module attributes such as comprehensibility or maintainability.

TABLE 7
Support for the Hypotheses Provided in This Case Study

Number	Hypothesis	Case study evidence?
1a	a small number of modules contain most of the faults discovered during pre-release testing	Yes - evidence of 20-60 rule
1b	if a small number of modules contain most of the faults discovered during pre-release testing then this is simply because those modules constitute most of the code size	No
2a	a small number of modules contain most of the operational faults	Yes - evidence of 20-80 rule
2b	if a small number of modules contain most of the operational faults then this is simply because those modules constitute most of the code size	No - strong evidence of a converse hypothesis
3	Higher incidence of faults in function testing (FT) implies higher incidence of faults in system testing	Weak support
4	Higher incidence of faults in all pre-release testing implies higher incidence of faults in post-release operation	No - strongly rejected
5a	Smaller modules are less likely to be failure prone than larger ones	No
5b	Size metrics (such as LOC) are good predictors of number of pre-release faults in a module	Weak support
5c	Size metrics (such as LOC) are good predictors of number of post-release faults in a module	No
5d	Size metrics (such as LOC) are good predictors of a module's (pre-release) fault-density	No
5e	Size metrics (such as LOC) are good predictors of a module's (post-release) fault-density	No
6	Complexity metrics are better predictors than simple size metrics of fault and failure-prone modules	No (for cyclomatic complexity), but some weak support for metrics based on SigFF
7	Fault densities at corresponding phases of testing and operation remain roughly constant between subsequent major releases of a software system	Yes
8	Software systems produced in similar environments have broadly similar fault densities at similar testing and operational phases	Yes

We investigated the extent to which benchmarking type data could provide insights into software quality. In testing Hypotheses 7 and 8, we showed that the fault densities are roughly constant between subsequent major releases and our data indicates that there are 10-30 times as many prerelease faults as postrelease faults. Even if readers are uninterested in the software engineering hypotheses (1-6), they will surely value the publication of these figures for future comparisons and benchmarking.

We believe that there are no "software engineering laws" as such, because it is always possible to construct a system in an environment which contradicts the law. For example, the studies summarized in [17] suggest that larger modules have a lower fault density than smaller ones. Apart from the fact that we found no clear evidence of this ourselves (Hypothesis 5) and also found weaknesses in the studies, it would be very dangerous to state this as a law of software engineering. You only need to change the amount of testing you do to "buck" this law. If you do not test or use a module, you will not observe faults or failures associated with it. Again, this is because the association between size and fault density is not a causal one. It is for this kind of reason that we recommend more complete models that enable us to augment the empirical observations with other explanatory factors, most notably, *testing effort* and *operational usage* (as discussed, for example, in [8] and [32]). In this sense, our results justify the recent work on building causal models of software quality using Bayesian Belief Networks rather than traditional statistical methods which are patently inappropriate for defects prediction [33]. In the case study systems, we did not have available (at the module level) either testing effort or operational usage data, but the company has since agreed to collect this data to help with future modeling.

In the case study system described in this paper, the data-collection activity is considered to be a part of routine

configuration management and quality assurance. We have used this data to shed light on a number of issues that are central to the software engineering discipline. If more companies shared this kind of data, the software engineering discipline could quickly establish the empirical and scientific basis that it so sorely lacks.

ACKNOWLEDGMENTS

This work was conducted while Norman Fenton was at the Centre for Software Reliability, City University. We are indebted to Martin Neil for his valuable input to this work and to Pierre-Jacques Courtois, Karama Kanoun, Jean-Claude Laprie, and Stuart Mitchell for their review comments. The anonymous TSE referees and the associate editor John Rushby, provided valuable comments which have certainly improved the paper. The work was supported, in part, by the EPSRC-funded project IMPRESS, the ESPRIT-funded projects DEVA and SERENE, the Swedish National Board for Industrial and Technical Development, and Ericsson Utvecklings AB.

REFERENCES

[1] E. Adams, "Optimizing Preventive Service of Software Products," *IBM Research J.,* vol. 28, no. 1, pp. 2–14, 1984.

[2] W.W. Agresti and W.M. Evanco, "Project Software Defects From Analyzing Ada Designs," *IEEE Trans. Software Eng.,* vol. 18, no. 11, pp. 988–997, Nov. 1992.

[3] V.R. Basili and B.T. Perricone, "Software Errors and Complexity: An Empirical Investigation," *Comm. ACM,* vol. 27, no. 1, pp. 42–52, 1984.

[4] D.W. Carman, A.A. Dolinsky, M.R. Lyu, and J.S. Yu, "Software Reliability Engineering Study of a Large-Scale Telecommunications System," *Proc. Sixth Int'l Symp. Software Reliability Eng.,* pp. 350–359, 1995.

[5] D.A. Christenson and S.T. Huang, "Estimating the Fault Content of Software Suing the Fix-on-Fix Model," *Bell Labs Technical J.,* vol. 1, no. 1, pp. 130–137, 1996.

[6] T.B. Compton and C. Withrow, "Prediction and Control of ADA Software Defects," *J. Systems Software,* vol. 12, pp. 199–207, 1990.

[7] M.K. Daskalantonakis, "A Practical View of Software Measurement and Implementation Experiences within Motorola," *IEEE Trans. Software Eng.,* vol. 18, no. 11, pp. 998–1,010, Nov. 1992.

[8] M. Dyer, *The Cleanroom Approach to Quality Software Development.* John Wiley & Sons, 1992.

[9] C. Ebert and T. Liedtke, "An Integrated Approach to Criticality Prediction," *Proc. Sixth Int'l Symp. Software Reliability Eng.,* pp. 14–23, 1995.

[10] S.G. Eick, C.R. Loader, M.D. Long, L.G. Votta, and S. Vanderweil, "Estimating Software Fault Content before Coding," *Proc. 14th Int'l Conf. Software Eng.,* pp. 59–65, 1992.

[11] N.E. Fenton, "Software Measurement: A Necessary Scientific Basis," *IEEE Trans Software Eng.,* vol. 20, no. 3, pp. 199–206, Mar. 1994.

[12] N.E. Fenton and B.A. Kitchenham, "Validating Software Measures," *J. Software Testing, Verification, and Reliability,* vol. 1, no. 2, pp. 27–42, 1991.

[13] N.E. Fenton and S.L. Pfleeger, *Software Metrics: A Rigorous and Practical Approach,* second ed., Int'l Thomson Computer Press, 1996.

[14] R. Gibson, *Managing Computer Projects.* London: Prentice Hall, 1992.

[15] R.B. Grady, *Practical Software Metrics for Project Management and Process Improvement.* Prentice Hall, 1992.

[16] L. Hatton, "Static Inspection: Tapping the Wheels of Software," *IEEE Trans. Software Eng.,* pp. 85–87, vol 21, no. 5, May 1995.

[17] L. Hatton, "Software Failures: Follies and Fallacies," *IEE Review,* vol. 43, no. 2, pp. 49–52, Mar. 1997.

[18] U. Heitkoetter, B. Helling, H. Nolte, and M. Kelly, "Design Metrics and Aids to Their Automatic Collection," *J. Information and SoftwareTech.,* vol. 32, no. 1, pp. 79–87, 1990.

[19] S. Henry and D. Kafura, "Software Structure Metrics Based on Information Flow," *IEEE Trans. Software Eng.,* vol. 7, no. 5, pp. 510–518, 1981.

[20] K. Kaaniche and K. Kanoun, "Reliability of a Telecommunications System," *Proc. Seventh Int'l Symp. Software Reliability Eng.,* pp. 207–212, 1996.

[21] K. Kaaniche, K. Kanoun, M. Cukier, and M.M. Bastos, "Software Reliability Analysis of Three Successive Generations of a Switching System," *Proc. First European Conf. Dependable Computing (EDCC-1),* pp. 473–490, 1994.

[22] K. Kanoun and T. Sabourin, "Software Dependability of a Telephone Switching System," *Proc. 17th IEEE Int'l Symp. Fault-Tolerant Computing (FTCS-17),* pp. 236–241, 1987.

[23] K. Kanoun, M. Kaaniche, and J.-C. Laprie, "Experience in Software Reliability: From Data Collection to Quantitative Evaluation," *Proc. Fourth Int'l Symp. Software Reliability Eng.,* pp. 234–245, 1993.

[24] G.Q. Kenney and M.A. Vouk, "Measuring the Field Quality of Wide-Distribution Commerical Software," *Proc. Third Int'l Symp Software Reliability Eng.,* pp. 351–357, 1992.

[25] T.M. Khoshgoftaar, E.B. Allen, K.S. Kalaichelvan, and N. Goel, "Early Quality Prediction: A Case Study in Telecommunications," *IEEE Trans. Software Eng.,* vol 13, no. 1, pp. 65–71, Jan. 1996.

[26] B.A. Kitchenham, A.P. Kitchenham, and J.P. Fellows, "The Effects of Inspections on Software Quality and Productivity," *ICL Technical J,* pp. 112–22, May 1986.

[27] B.A. Kitchenham, L.M. Pickard, and S.J. Linkman, "An Evaluation of Some Design Metrics," *Software Eng. J.,* vol 5, no. 1, pp. 50–58, 1990.

[28] Y. Levendel, "Reliability Analysis of Large Software Systems: Defects Data Modelling," *IEEE Trans. Software Eng.,* vol. 16, no. 2, pp. 141–152, Feb. 1990.

[29] T. McCabe, "A Software Complexity Measure," *IEEE Trans. Software Eng.,* vol. 2, no. 4, pp. 308–320, 1976.

[30] K.-H. Moller and D. Paulish, "An Empirical Investigation of Software Fault Distribution," *Software Quality Assurance and Measurement,* N.E. Fenton, R.W. Whitty, and Y. Iizuka, eds., pp. 242–253, Int'l Thomson Computer Press, 1995.

[31] J.C. Munson and T.M. Khoshgoftaar, "The Detection of Fault-Prone Programs," *IEEE Trans. Software Eng.,* vol. 18, no. 5, pp. 423–433, May 1992.

[32] J. Musa, "Operational Profiles in Software Reliability Engineering," *IEEE Trans. Software Eng.,* vol. 10, no. 2, pp. 24–32, Feb. 1993.

[33] M. Neil and N.E. Fenton, "Predicting Software Quality Using Bayesian Belief Networks," *Proc 21st Ann. Software Eng. Workshop,* pp. 217–230, Dec. 1996.

[34] N. Ohlsson, "Predicting Error-Prone Software Modules in Telephone Switches." Masters thesis, Linköping Univ., Sweden, 1993.

[35] N. Ohlsson and H. Alberg, "Predicting Error-Prone Software Modules in Telephone Switches," *IEEE Trans. Software Eng.,* vol. 22, no. 12, pp. 886–894, Dec. 1996.

[36] N. Ohlsson, M. Zhao, and M. Helander, "Application of Multivariate Analysis for Software Fault Prediction," *Software Quality J.,* vol. 7, no. 1, pp. 51–66, 1998.

[37] S.L. Pfleeger and L. Hatton, "Investigating the Influence of Formal Methods," *IEEE Computer,* vol. 30, no. 2, pp. 33–43, Feb. 1997.

[38] V.Y. Shen, T. Yu, S.M. Thebaut, and L.R. Paulsen, "Identifying Error-Prone Software—An Empirical Study," *IEEE Trans. Software Eng.,* vol 11, no. 4, pp. 317–323, Apr. 1985.

[39] K.J. Turner, *Using Formal Description Techniques—An Introduction to ESTELLE, LOTOS and SDL.* John Wiley & Sons, 1993.

[40] I. Vessey and R. Weber, "Research on Structured Programming: An Empiricist's Evaluation," *IEEE Trans. Software Eng.,* vol. 10, no. 7, pp. 397–407, July 1984.

[41] K. Yasuda and K. Koga, "Product Development and Quality in the Software Factory," *Software Quality Assurance and Metrics: A Worldwide Prespective,* N.E. Fenton, R.W. Whitty, and Y. Iizuka eds., pp. 195–205, Int'l Thomson Press, 1995.

[42] T.J. Yu, V.Y. Shen, and H.E. Dunsmore, "An Analysis of Several Software Defect Models," *IEEE Trans. Software Eng.,* vol. 14, no. 9, pp. 1,261–1,270, Sept. 1988.

[43] H. Zuse, *Software Complexity: Measures and Methods.* Berlin: De Gruyter, 1991.

[44] J.M. Juran, F.M. Gryna, Jr., and F.M. Bingham, *Quality Control Handbook.* Third edition, McGraw Hill, New York, 1979.
[45] G.G. Schulmeyer and J.I. McManus, *Handbook for Software Quality Assurance.* G.G. Schulmeyer and J.I. McManus eds., Van Nostrand Rheinhold, New York, 1987.
[46] N.E. Fenton and M. Neil, "A Critique of Software Defect Prediction Models," *Trans. Software Eng.,* vol. 25, no. 5, pp. 675–689, May 1999.

Norman Fenton is a professor of computing science and is head of RADAR (Risk Assessment and Decision Analysis Research) at the Computer Science Department, Faculty of Informatics and Mathematical Sciences, Queen Mary and Westfield College, London, he is also director of Agena Ltd. He was formerly with the Centre for Software Reliability, City University (London). His research interests include software metrics, empirical software engineering, safety critical systems, and formal development methods. The focus of his current work is on applications of Bayesian nets; these applications include critical systems' assessment, vehicle reliability prediction, and software quality assessment. He is a member of the IEEE Computer Society.

Niclas Ohlsson received his MS (1993) and PhD (1998) in computer science from Linköping University, Sweden. His research has been carried out in close collaboration with Ericsson UAB, SAAB Aerospace, and the Swedish Defense Material Administration (FMV). He joined GratisTel International in August 1998, where he is the technical director. His research interests include software quality engineering, continuous process improvement, software metrics and learning organizations. He is a member of the IEEE Computer Society.