Quantitative Analysis of Faults and Failures in a Complex Software System

Norman E Fenton
Centre for Software Reliability
City University

Niclas Ohlsson
Dept Computer and Information Sciences
University of Linköping

Version 2.0, 22 January 1998

Abstract 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 very strong evidence that a small number of modules contain most of the faults discovered in pre-release 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 pre-release 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 post release faults: those modules which are the most fault-prone pre-release are among the least fault-prone post-release, while conversely the modules which are most fault-prone post release are among the least fault-prone pre-release. 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 important data-points in building up an empirical picture of the software development process. However, we believe that even the very 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.

1 Introduction

Despite some heroic efforts from a small number of research centres and individuals (see, for example [Carman et al 1995], [Kaaniche and Kanoun 1996], [Khoshgoftaar et al 1996], [Ohlsson N and Alberg 1996], [Shen et al 1985]) there continues to be a dearth of published empirical data relating to the quality and reliability of realistic commercial software systems. Two of the best and most important studies [Adams 1984] and [Basili and Perricone 1984] are now over 12 years old. Adams' study 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 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 pre-defined phase of testing or operation) divided by a measure of module size (normally KLOC). While the fault density measure has numerous weaknesses as a quality measure (see [Fenton and Pfleeger 1996] 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 [Moeller and Paulish 1995]. 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’ [Hatton 1997].

Thus the various empirical studies have thrown up results which are counter-intuitive to 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 [Fenton *et al* 1994] we commented on the almost total absence of empirical research on evaluating the effectiveness of different software development and testing methods. There also continues to be an almost total absence of published benchmarking data.

In this paper we hope to provide a 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. 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 following hypotheses:

- Hypotheses relating to the Pareto principle of distribution of faults and failures

1a) a small number of modules contain most of the faults discovered during pre-release testing;

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

2a) a small number of modules contain the faults that cause most failures

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.

- 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 pre-release testing implies higher incidence of failures in operation.

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

- Hypotheses about metrics for fault prediction

  5) Simple size metrics (such as 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 very strong evidence for and against some of the above hypotheses and also explain how some previous studies that have looked at these hypotheses are flawed. Hypotheses 1a and 2a are strongly supported, while 1b and 2b are strongly rejected. Hypothesis 3 is weakly supported, while curiously hypothesis 4 is strongly rejected. Hypothesis 5 is partly supported, but hypotheses 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 8 can only be tested properly once other organisations publish analogous results.

We discuss the results in more depth in Section 4.

2 The basic data

The data presented in this paper is based on two major consecutive releases of a large legacy project developing telecommunication switching systems. 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. The modules ranged in size from approximately 1000 to 6000 LOC (as shown in Table 1). Both releases were approximately the same total system size.

1 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

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 phases:

- function test (FT)
- system test (ST)
- first 26 weeks at a number of site tests (SI)
- first year (approx) operation (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 organisation develop 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.* We will refer to the combination of SI and OP faults collectively as *operational faults*. We shall also refer at times to *failures*. Formally, a failure is an observed deviation of the operational system behaviour from specified or expected behaviour. 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:

a) the fault had already been corrected;

b) the fault will be corrected;

c) the fault requires no action (i.e. not treated as a fault);

d) the fault was due to installation problems.

In this paper we have only considered faults classified as *b*. 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.

	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

Table 2. Distribution of faults per testing phase

2.2 Independent variables

Various metrics were collected for each module. These included:

- Lines of code (LOC) as the main size measure
- McCabe's cyclomatic complexity.
- Various metrics based on communication (modelled with signals) between modules and within a module During the specification phase, the number of new and modified signals (signals are similar to messages) for each module were specified. Most notably, the metric *SigFF* is the count of the number of new and modified signals. This metric was also used as a measure of interphase complexity. [Ohlsson and Alberg, 1996] provides full details of these metrics and their computation.

The complexity metrics were collected automatically from the actual design documents using a tool, ERIMET [Ohlsson, 1993]. 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 [Turner, 1993]. The metrics are extracted direct from the FDL-graphs. The fact that the metrics were computed from artefacts 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 [Heitkoetter et al 1990]. However, there have been very few published attempts to do so. [Kitchenham et al, 1990] reported on using design metrics, based on Henry and Kafura's information and flow metrics

[1981 and 1984], for outlier analysis. [Khoshgoftaar et al, 1996] used a subset of metrics that "could be collected from design documentation", but the metrics were extracted from the code.

Numerous studies, such as [Ebert and Liedtke, 1995]; and [Munson and Khoshgoftaar, 1992] have reported using metrics extracted from source code, but few have reported promising prediction results based on design metrics.

3 The hypotheses tested and results

Since the data were collected and analysed 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% of the faults are contained in 20% of the modules. We show that there is strong 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;

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

The main part of the total cost of quality deficiency is often found to be caused by very few faults or fault types [Bergman and Klefsjo 1991]. The Pareto principle [Juran 1964], also called the 20-80 rule, summarises 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 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 [1984] demonstrated that a small number of faults were responsible for a large number of failures. [Munson *et al* 1992] motivated their

discriminative analysis by referring to the 20-80 rule, even though their data demonstrated a 20-65 rule. [Zuse 1991] used Pareto techniques to identify the most common types of faults found during function testing. Finally, [Schulmeyer and MacManus 1987] 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 pre-release testing (phases FT and ST);

Hypothesis 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.

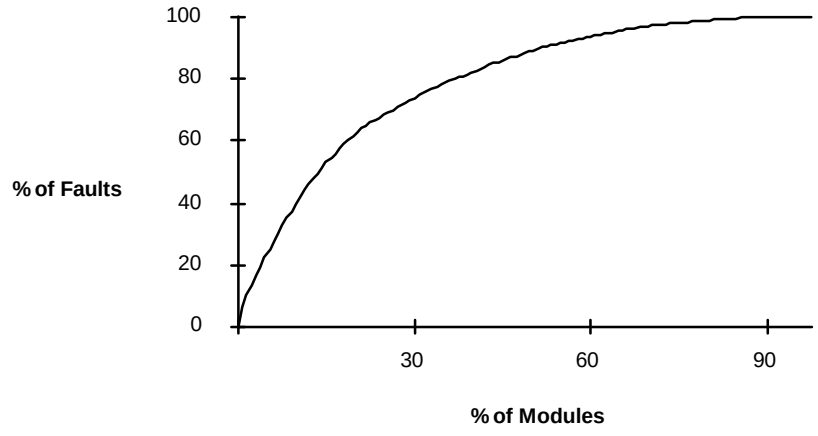
Hypothesis 2a: a small number of modules contain most of the operational faults (meaning failures as we have defined them above observed in phases SI and OP);

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: a small number of modules contain most of the faults discovered during testing (phases FT and ST)

Figure 1 illustrates that 20% of the modules were responsible for nearly 60% 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 [Ohlsson et al 1996]. This, together with other results such as [Munson *et al* 1992], provides very strong support for hypothesis 1a), and even suggests a specific Pareto distribution in the area of 20-60. This 20-60 finding is not as strong as the one observed by [Compton and Withrow, 1990] (they found that 12% of the modules, referred to as packages, accounted for 75% of all the faults during system integration and test), but is nevertheless important.



1

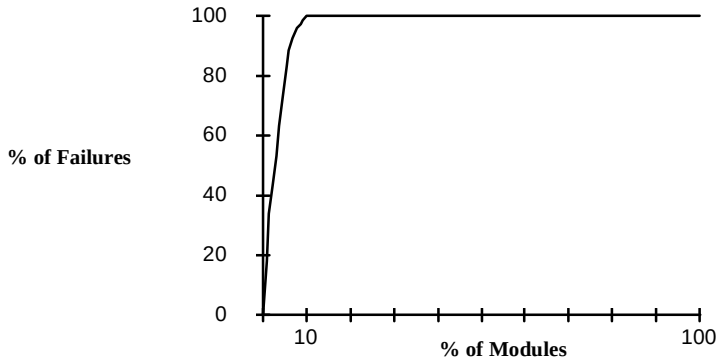
Figure 1: **Pareto diagram showing percentage of modules versus percentage of faults for Release *n***

3.1.2 Hypothesis 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.

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, [Compton and Withdraw, 1990] found that the 12% of modules accounting for 75% of the faults accounted for 63% of the LOC. In our study we found no evidence to support hypotheses 1b. For release *n*, the 20% of the modules which account for 60% of the faults (discussed in hypothesis 1a) actually make up just 30% of the system size. The result for release *n+1* was almost identical.

3.1.3 Hypothesis 2a: a small number of modules contain most of the operational faults (meaning failures as we have defined them above, namely phases CU and OP)

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



1

Figure 2: Pareto diagram showing percentage of modules versus percentage of failures for Release *n*

3.1.4 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.

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 2, but we discovered very strong evidence in favour of a converse hypothesis:

most operational faults are caused by faults in a small proportion of the code

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

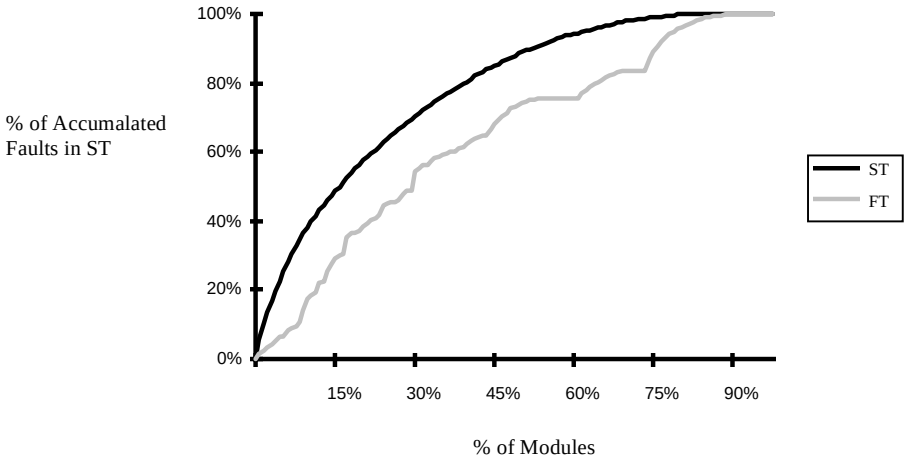
Hypothesis 3: Higher incidence of faults in function testing (FT) implies higher incidence of faults in system testing (ST)

Hypothesis 4: Higher incidence of faults in all pre-release testing (FT and ST) implies higher incidence of faults in post-release operation (SI and OP).

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

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

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



1

Figure 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*.

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% of the most fault-prone modules in system test are responsible for 38% of the faults in system test, but 10% of the most fault-prone modules in function test is only responsible for 17% of the faults in system test. This is persistent up to 75% of the modules. This means that nearly 20% of the faults in system test need to be explained in another way. The same pattern was found when using normalised data (faults/LOC) instead of absolute, even though the percentages were general lower and the prediction a bit poorer.

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

- 50% of the faults in system test occurred in modules which were responsible for 25% of the faults in function test
- 10% of the most fault-prone modules in system test are responsible for 46% of the faults in system test, but 10% of the most fault-prone modules in function test is only responsible for 24% of the faults in system test.

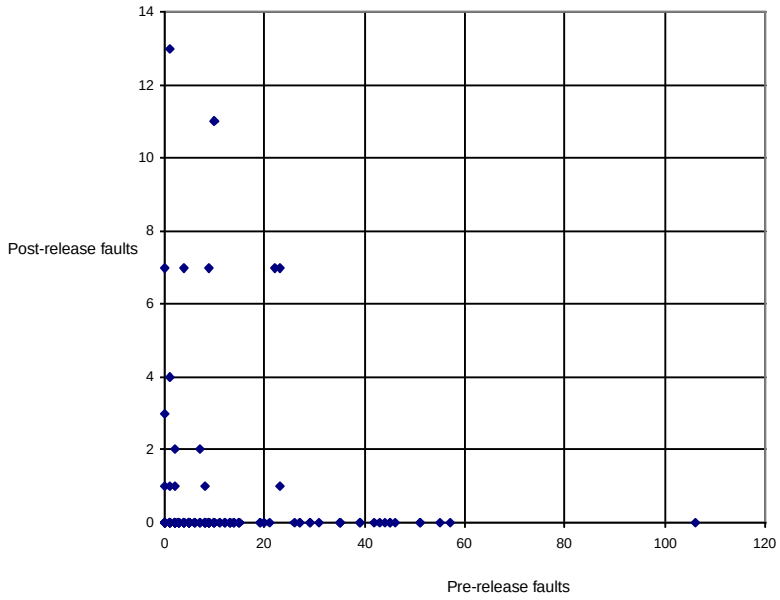
These results and also when using normalised data instead of absolute are very similar to the result in release *n*.

3.2.2 Hypothesis 4: Higher incidence of faults in all pre-release testing (FT and ST) implies higher incidence of faults in post-release operation (SI and OP).

The rationale behind hypothesis 4 is that the relatively small proportion of modules in a system that account for most of the faults are likely to be fault-prone both pre- and post release. 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, 1990] have found as much as six times greater post delivery defect density when analysing modules with faults discovered prior to delivery.

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 again there is strong evidence to support a converse hypothesis. In both release *n* and release *n+1* almost all of the faults discovered in pre-release testing appear in modules which subsequently reveal almost no operation faults. Specifically, we found:

- In release *n* (see Figure 4), 93% of faults in pre-release testing occur in modules which have NO subsequent operational faults (of which there were 75 in total). Thus 100% of the 75 failures in operation occur in modules which account for just 7% of the faults discovered in pre-release testing.



1

Figure 4: Scatter plot of pre-release faults against post-release faults for version *n* (each dot represents a module)

- In release *n+1* we observed a much greater number of operational faults, but a similar phenomenon to that of release *n* (see Figure 5). Some 77% of pre-release faults occur in modules which have NO post-release faults. Thus 100% of the 366 failures in operation occur in modules which account for just 23% of the faults discovered in function and system test.

These remarkable results are also exciting because they are closely related to the Adams' phenomenon. The results have major ramifications for one of the most commonly used software measures, *fault density*. Specifically it appears that modules with high fault density pre-release are likely to have low fault-density post-release, and vice versa. We discuss the implications at length in Section 4.

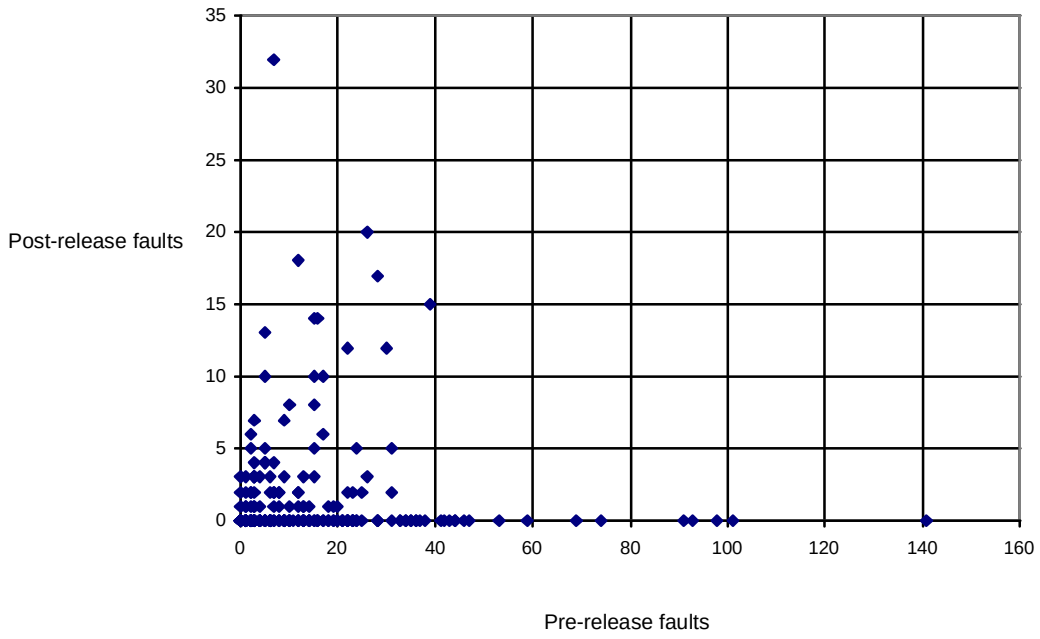


Figure 5: Scatter plot of pre-release faults against post-release faults for version *n+1* (each dot represents a module)

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 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 [Fenton and Pfleeger 1997]. 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: Size metrics (such as LOC) are good predictors of fault and failure prone modules.

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 pre-release faults in a module

Hypothesis 5c: Size metrics (such as LOC) are good predictors of number of post-release faults in a module

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

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

Hypothesis 5*a* underpins, in many respects, the principles behind most modern programming methods, such as modular, structured, and objected 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 organisations.

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 [1984] reported that fault density appeared to *decrease* with module size. Their explanation to this was the large number of interface faults spread equally across all modules. The relatively high proportion of small modules were also offered as an explanation. Other authors, such as [Moeller and Paulish 1995] 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 begin our results with a replication of the key part of the [Basili and Pericone 1984] study. Table 3 (which compare 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 data set analysed in this paper has, in comparison with [Basili and Pericone 1984] 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 [Basili and Pericone 1984], but we do not believe this introduces any bias.

	Release n			Release n+1			
Fault	Mod	New	Percent modified modules	Mod	New	Splitted	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

Table 3. Number of Modules Affected by a fault for Release n (140 modules, 1815 Faults) and Release *n+1* (246 modules, 3795 faults).

The scatter plots Figure 6, for lines of code versus the number of pre- and post-release faults does not reveal any strong evidence of trends for release *n+1*. Neither could any strong trends be observed when line of code versus the total number of faults were graphed, Figure 7. The results for release *n* were reasonably similar.

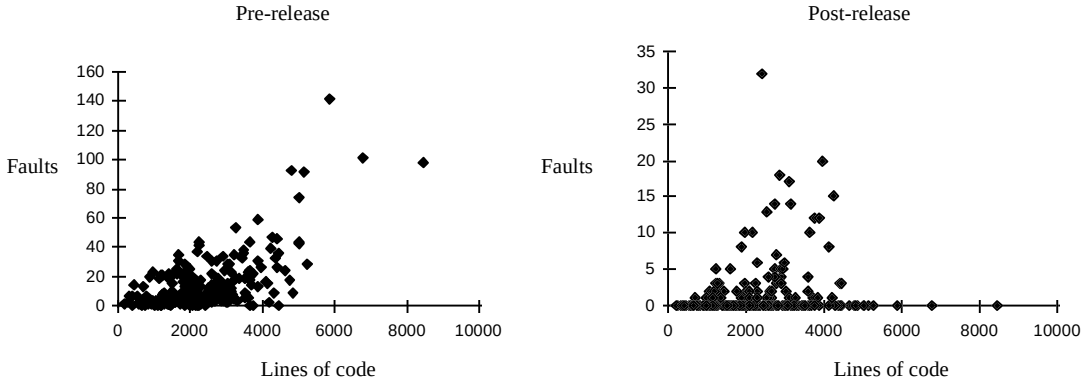


Figure 6: Scatterplots of LOC against pre- and post-release faults for release *n+1* (each dot represents a module).

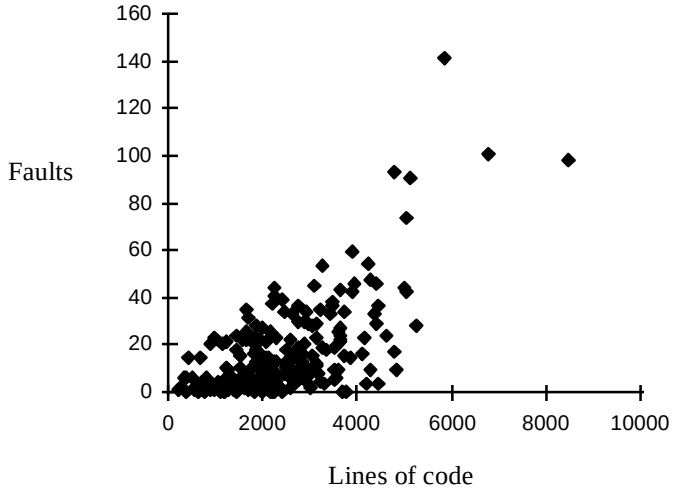


Figure 7: Scatterplots of LOC against all faults for release *n+1* (each dot represents a module).

When Basili and Pericone could not see any trend they calculated the number of faults per 1000 executable lines of code. Table 4 (which compares with table VII in [Basili and Pericone 1984]) shows these results for our study.

	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

Table 4. Faults/1000 Lines of code release *n* and *n+1*.

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 [Basili and Perricone 1984] is highly misleading. What Basili and Pericone failed to show was a simple plot of fault density against module size, as we have done in Figure 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. Clearly other explanatory factors, such as design, inspection and testing effort per module, will be more important.

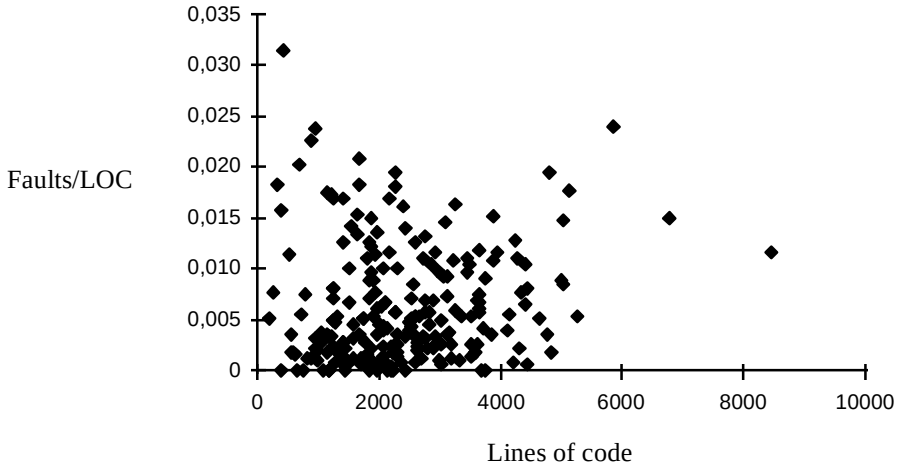


Figure 8: Scatter plot of module fault density against size for release *n+1*

The scatter plots assumes that the data belong to an interval or ratio scale. From a prediction perspective it is not always necessary. In fact, a number of studies are built on the Pareto principle, which often only require that we have ordinal data. In the tests of hypothesis above we have used a technique that is based on ordinal data, called Alberg diagrams [Ohlsson and Alberg 1996], to evaluate the independent variables' ability to rank the dependent variable. The LOC ranking ability is assessed in Figure 9. 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.

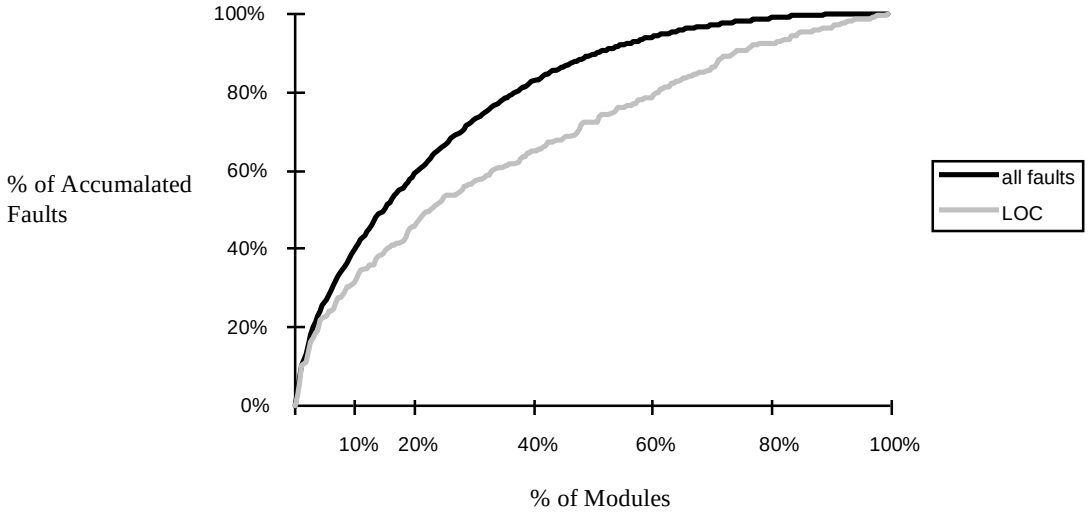


Figure 9. Accumulated percentage of the absolute number of all faults when modules are ordered with respect to LOC for release *n+1*.

3.3.2 Hypothesis 6: Complexity metrics are better predictors than simple size metrics of fault and failure-prone modules

'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 [McCabe, 1976], but there have in fact been many dozens that have been published [Zuse 1991]. The details, and also the limitations of complexity metrics, have been extensively documented (see [Fenton and Pfleeger 1996]) and we do not wish to re-visit those issues here. What we are concerned with here is the underlying assumption that 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:

> 'There is a clear intuitive basis for believing that complex programs have more faults in them than simple programs', [Munson and Khosghoftaar, 1992]

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). We have already seen, in section 3.3.1, that size is a reasonable predictor of number of faults (although not of fault density). We now investigate the case of complexity metrics such as the cyclomatic number.

We demonstrated in testing the last hypothesis the problem with comparing average figures for different size intervals. Instead of replicating the relevant analysis in [Basili and Pericone 1984] by calculating the average cyclomatic number for each module size class, and than plotting the results we just generated scatter plots and Alberg diagrams.

When the cyclomatic complexity and the pre- and post-release faults were graphed for release *n+1* (Figure 10) we observed a number of interesting trends. The most complex modules appear to be more fault-prone in pre-release, but appear to have nearly no faults in post-release. The most fault-prone modules in post-release 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. Analysing in retrospect the earlier graphs for size versus faults reveal a similar pattern.

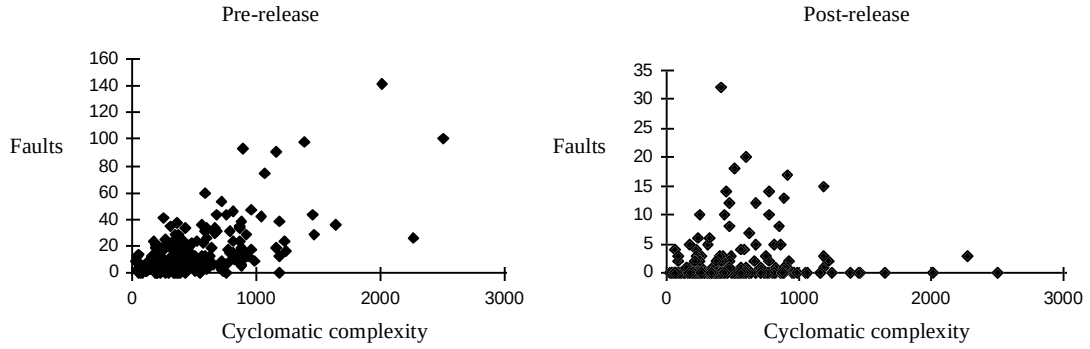


Figure 10: Scatterplots of cyclomatic complexity against number of pre-and post-release faults for release *n+1* (each dot represents a module).

The scatter plot for the cyclomatic complexity and the total number of faults (Figure 11) shows again some small indication of correlation. The Alberg diagrams were similar as when size was used.

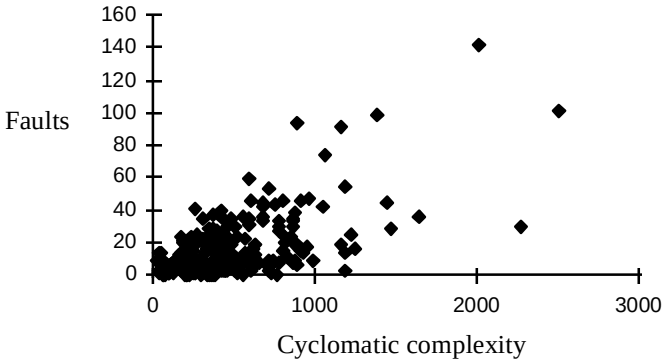


Figure 11: Scatterplot of cyclomatic complexity against all faults for release *n+1* (each dot represents a module).

To explore the relations further the scatter plots were also graphed with normalised data (Figure 12). The result showed even more clearly that the most-fault prone modules in pre-release have nearly no post-release faults.

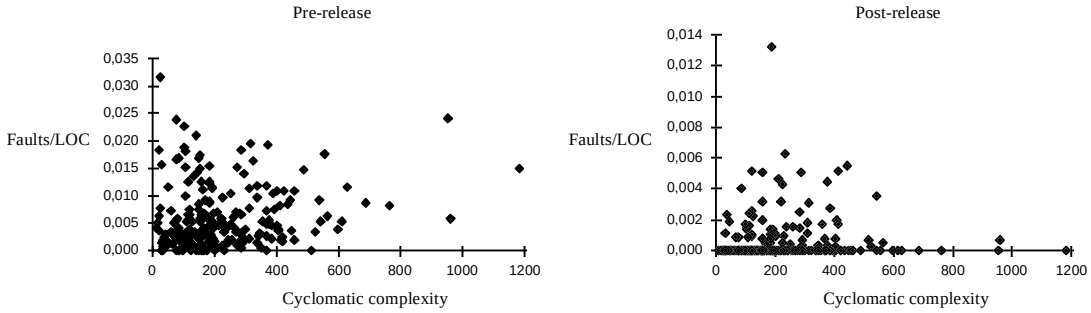


Figure 12: Scatterplots of cyclomatic complexity against fault density (pre-and post-release) for release *n+1* (each dot represents a module).

In order to determine whether or not large modules were less dense or complex than smaller modules [Basili and Perricone, 1984] plotted the cyclomatic complexity versus module size. Following the same pattern in earlier analysis they failed to see any trends, and therefore they analysed the relation by grouping modules according to size. As illustrated above this can be very misleading. Instead we graphed scatter plots of the relation and calculated the correlation (Figure 13).

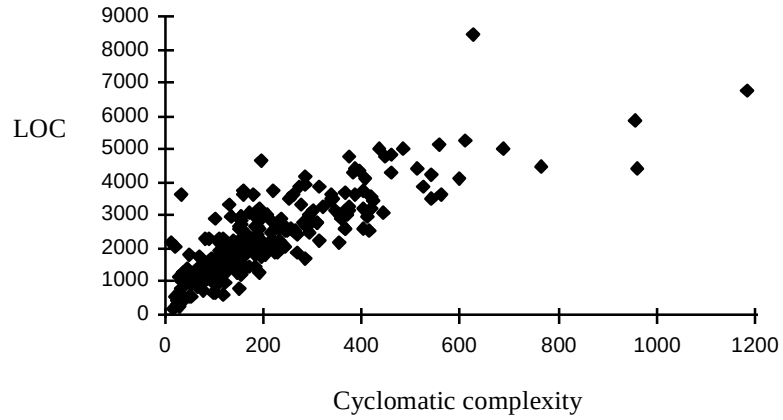


Figure 13: Complexity versus Module Size

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

Earlier studies [Ohlsson and Alberg, 1996] 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 *SigFF* measure instead of cyclomatic complexity.

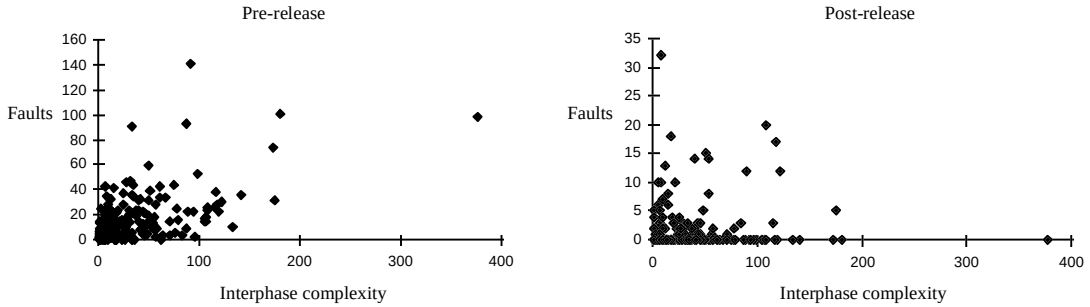


Figure 14: Scatterplots of SigFF against number of pre-and post-release faults for release *n+1* (each dot represents a module).

The scatterplots using absolute numbers (Figure 14), or normalised data did not indicate any new trends. In earlier work the product of cyclomatic complexity and *SigFF* was shown to be a good predictor of fault-proneness. To evaluate *CC*SigFF* predictability the Alberg diagram was graphed (Figure 15). The combined metrics appear to be better than both SigFF and Cyclomatic Complexity on there own, and also better than the size metric.

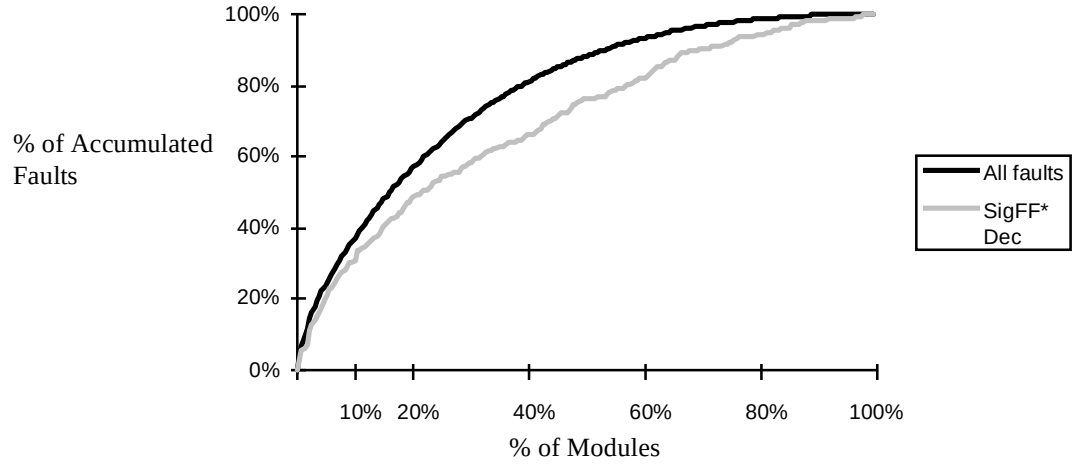


Figure 15. Accumulated percentage of the absolute number of all faults when modules are ordered with respect to LOC for release *n+1*.

The above results do not paint a very glowing report of the usefulness of complexity metrics, but it can be argued that 'being a good predictor of fault density' is not an appropriate validation criteria for complexity metrics. This is discussed in section 4. Nevertheless there are some positive aspects. The combined metric *CC*SigFF* is again shown to be a reasonable predictor of fault-prone modules. Also, measures like *SigFF* 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 publicising the kind of data discussed in this paper is to enable both intra- and inter-company comparisons. Despite the incredibly vast

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, almost nothing has ever been published. The 'grey' literature (as referenced, for example, in [Pfleeger and Hatton 1997] seems to suggest some crude (but unsubstantiated guidelines) such as the following for fault density in first 12 months of typical operational use:

- less that 1 fault per KLOC is very good (and typically only achieved by companies using state-of-the-art development and testing methods)
- between 4 to 8 faults per KLOC is typical
- greater than 12 faults per KLOC is bad

When pre-release 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 pre-release 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 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

since we have data on successive releases. The results we present, being based only on one system, represents just a single data-point, but nevertheless we 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 Fault densities at corresponding phases of testing and operation remain roughly constant between subsequent major releases of a software system

	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 5: Fault densities at the four phases of testing and operation

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. This has interesting implications for the software process improvement movement, as epitomised by the Capability Maturity Model CMM.

A general assumption of CMM is that a stable and repeatable process is a necessary pre-requisite for continuous process improvement. For an immature organisation (below level 3) it is assumed to take many years to reach such a level. In CMM's terminology companies do not have the kind of stable and repeatable process indicated in the above figures until they are at level 3. Yet, like almost every software producing organisation in the world, the organisation in this case study project is *not* at level 3. The results reflects a stability and repeatability that according to CMM should not be the case. At such we question the CMM's underlying assumption about what constitutes an organisation that should have a stable and repeatable process.

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

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) pre-release fault density; and 2) post-release fault density. First, we can compare the two results of the two separate releases in the cases study (Table 6).

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

Table 6: Fault densities pre-and post-release for the case study system

The overall fault densities are similar to those reported for a range of systems in [Hatton 1995], while [Agresti and Evanco, 1992] reported similar ball-park figures in a study of Ada programs, 3.0 to 5.5 faults/KLOC. The post-release fault densities seem to be roughly in line of those reported studies of *best* practice.

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

Of the few published studies that reveal the difference between pre- and post-release fault density, [Pfleeger and Hatton, 1997] also report 10 times as many faults in pre-release (although the overall fault density is lower. [Kitchenham et al 1986] reports a higher ratio of pre-release to post-release. Their study was an investigation into the impact of inspections; combining the inspected and non-inspected code together reveals a pre-release fault density

of approx 16 per KLOC and a post-release fault density of approximately 0.3 per KLOC. However, it is likely that the operational time here was not as long.

Thus, from the small amount of evidence we conclude that there appears to be 10-30 times as many faults pre-release as post release.

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 control for software development. For example, this is the basis for the software factory approach proposed by Japanese companies such as Hitachi [Yasuda and Koga 1995] 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 come from just two releases of a major system developed by a single organisation. It may therefore be tempting for observers to dismiss their relevance for the broader software engineering community. Such an attitude would be dangerous given the rigour and extensiveness of the data-collection, and also the strength of some of the observations.

The evidence we found in support of the two Pareto principles 1a) and 2a) is the least surprising. It does seem to be inevitable that a small number of the modules in a system will contain a large proportion of the pre-release faults and that a small proportion of the modules will contain a large proportion of the post-release faults. However, the popularly believed explanations for these two phenomena appear to be quite wrong:

- 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 behaviour (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 pre-release are going to be roughly the same set of modules that are especially fault-prone post-release (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 is a very important observation. 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 as 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*.

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 *post-release*. Yet we have found that there is no relationship between the modules which are fault-prone pre-release and the modules which are fault-prone post-release. Most previous 'validation' studies of complexity metrics have deemed a metric 'valid' if it correlates with the (pre-release) 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 pre-release 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 SigFF 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 [Fenton and Pfleeger 1996], 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.

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 pre-release faults as post-release 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 summarised in [Hatton 1997] 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*. 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.[Neil and Fenton 1996].

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.

Acknowledgements

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 valuable review comments. 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

[Adams 1984] Adams E, 'Optimizing preventive service of software products', IBM Research Journal, 28(1), 2-14, 1984.

[Agresti and Evanco 1992] Agresti, W. W. and Evanco, W. M., *Project Software Defects From Analyzing Ada Designs*, IEEE Transactions on Software Engineering, 18(11):988-997, 1992.

[Basili and Perricone 1984] Basili VR and Perricone BT, 'Software Errors and Complexity: An Empirical Investigation', Communications of the ACM 27(1), pp. 42-52, 1984.

[Carman et al 1995] Carman DW, Dolinsky AA, Lyu MR, Yu JS, 'Software reliability engineering study of a large-scale telecommunications system', in 6th Int Symp on Software reliability Engineering, Toulouse, France, 350-359, 1995.

[Christenson and Huang 1996] Christenson DA and Huang ST, Estimating the fault content of software suing the fix-on-fix model, Bell Labs Tech J, 1(1), 130-137, 1996.

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

[Ebert and Liedtke 1995] Ebert, C. and Liedtke, T. (1995). An integrated approach to criticality prediction. In The Sixth International Symposium on Software Reliability Engineering, pages 14-23.

{Eick et al 1992] Eick SG, Loader CR, Long MD. Votta LG, Vanderweil S, Estimating software fault content before coding, Proc 14th International Conf Software Engineering, Melbourne, Australia, 59-65, 1992.

[Fenton 1994] Fenton NE, 'Software measurement: a necessary scientific basis', IEEE Trans Software Eng 20 (3), 199-206, 1994.

[Fenton and Pfleeger 1996] Fenton NE and Pfleeger SL, Software Metrics: A Rigorous and Practical Approach (2nd Edition), International Thomson Computer Press, 1996.

[Hatton 1995] Hatton L, Static inspection: tapping the wheels of software, IEEE Software, May, 85-87, 1995.

[Hatton 1997] Hatton L, 'Software failures: follies and fallacies', IEE Review 43(2), 49-52, March, 1997.

[Heitkoetter et al 1990] Heitkoetter U, Helling B, Nolte H, Kelly M, , Design metrics and aids to their automatic collection, J Inf & SoftwareTech. 32(1), 79-87., 1990.

[Henry and Kafura 1981] Henry, S. and Kafura, D. (1981). Software structure metrics based on information flow. IEEE Transactions on Software Engineering, 7(5):510-518.

[Kaaniche and Kanoun 1996] Kaaniche K, Kanoun K, Reliability of a Telecommunications System, in 7th Int. Symp. on Software Reliability Engineering, pp. 207-212, White Plains, NY, USA, 1996.

[Kaaniche et al 1994] Kaaniche K, Kanoun K, Cukier M, and Bastos Martini M, Software Reliability Analysis of Three Successive Generations of a Switching System, in First European Conference on Dependable Computing (EDCC-1), pp. 473-490, Berlin, Germany, 1994.

[Kanoun and Sabourin 1987] Kanoun K and Sabourin T, Software Dependability of a Telephone Switching System, in 17th IEEE Int Symp. on Fault-Tolerant Computing (FTCS-17), pp. 236-241, Pittsburgh, PA, USA, 1987.

[Kanoun et al 1993] Kanoun K, Kaaniche M and Laprie J.-C., Experience in Software Reliability: From Data Collection to Quantitative Evaluation, in 4th Int. Symp. on Software Reliability Engineering, pp.. 234-245, Denver, CO, USA, 1993.

[Khoshgoftaar et al 1996] Khoshgoftaar, T. M., Allen, E. B., Kalaichelvan, K. S., and Goel, N. Early quality prediction: a case study in telecommunications. IEEE Software, 13(1):65-71 1996

[Kitchenham et al 1986] Kitchenham BA, Kitchenham AP, Fellows JP, The effects of inspections on software quality and productivity, ICL Tech J, 112-22, May, 1986.

[Kitchenham et al 1990] Kitchenham, B. A., Pickard, L. M., and Linkman, S. J. (1990). An evaluation of some design metrics. Software Engineering Journal, 5(1):50-58.

[McCabe 1976] McCabe T, ‘A Software Complexity Measure’, IEEE Trans. Software Engineering SE-2(4), 308-320, 1976.

[Moller and Paulish 1995] Moller K-H and Paulish D, ‘An empirical investigation of software fault distribution’, in ' Software Quality Assurance and Measurement' (Eds Fenton NE, Whitty RW, Iizuka Y), International Thomson Computer Press, 242-253, 1995.

[Munson and Khoshgoftaar 1992] Munson JC, and Khoshgoftaar TM, ‘The detection of fault-prone programs’, IEEE Transactions on Software Engineering,18(5), 423-433, 1992.

[Neil and Fenton 1996] Neil M and Fenton NE, Predicting software quality using Bayesian belief networks, Proc 21st Annual Software Eng Workshop, NASA Goddard Space Flight Centre, 217-230, Dec, 1996.

[Ohlsson and Alberg 1996] Ohlsson N and Alberg H, ‘Predicting error-prone software modules in telephone switches’, IEEE Transactions on Software Engineering, 22(12), 886-894, 1996.

[Ohlsson 1993] Ohlsson, N. (1993). Predicting error-prone software modules in telephone switches. Master' s thesis, Department of computer and information science, Linköping University.

[Pfleeger and Hatton 1997] Pfleeger S.L. and Hatton L, Invesitgating the influence of formal methods, IEEE Computer, 30(2), 33-43, 1997.

[Shen et al 1985] Shen VY, Yu T, Thebaut SM, Paulsen LR, Identifying error-prone software -- an empirical study, IEEE Trans Soft Eng SE-11(4), 317-323, 1985.

[Turner 1993] Turner, K. J., editor (1993). Using formal description techniques - An introduction to ESTELLE, LOTOS and SDL. John Wiley & Sons.

[Vessey and Weber 1984] Vessey I and Weber R, ‘Research on structured programming: an empiricist's evaluation’, IEEE Trans Software Eng, 10, 397-407, July, 1984.

[Yasuda K and Koga 1995] Yasuda K and Koga K, Product development and quality in the software factory, in ' Software Quality Assurance and metrics: AWorldwide prespective' (Eds: Fenton NE, Whitty RW, Iizuka Y, International Thomson Press, 195-205, 1995.

[Yu et al 1988] Yu TJ, Shen VY, Dunsmore HE, An analysis of several software defect models, IEEE Transactions on Software Engineering, 14(9), 1261-1270, 1988.

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