The Distribution of Faults in a Large Industrial Software System

Thomas J. Ostrand
ostrand@research.att.com
AT&T Labs - Research
180 Park Avenue
Florham Park, NJ 07932

Elaine J. Weyuker
weyuker@research.att.com
AT&T Labs - Research
180 Park Avenue
Florham Park, NJ 07932

ABSTRACT

A case study is presented using thirteen releases of a large industrial inventory tracking system. Several types of questions are addressed in this study. The first involved examining how faults are distributed over the different files. This included making a distinction between the release during which they were discovered, the lifecycle stage at which they were first detected, and the severity of the fault. The second category of questions we considered involved studying how the size of modules affected their fault density. This included looking at questions like whether or not files with high fault densities at early stages of the lifecycle also had high fault densities during later stages. A third type of question we considered was whether files that contained large numbers of faults during early stages of development, also had large numbers of faults during later stages, and whether faultiness persisted from release to release. Finally, we examined whether newly written files were more fault-prone than ones that were written for earlier releases of the product. The ultimate goal of this study is to help identify characteristics of files that can be used as predictors of fault-proneness, thereby helping organizations determine how best to use their testing resources.

Keywords: Software Faults, Fault-prone, Pareto, Empirical Study, Software Testing

1. INTRODUCTION

In spite of its importance, there have been relatively few empirical studies published that investigate issues relating to the dependability of software. Of those, only a small number have been done using industrial software systems of large size and complexity. The ones most relevant to the work described in this paper include [1, 3, 6] Although it is possible that studies of this sort are being performed, the primary evidence that we would see, published papers, are rare. We believe that we do not see many large empirical studies in the literature because they are not being done, and the reasons are straightforward:

- It is difficult to locate and gain access to these large systems.
- It is very time consuming, and therefore expensive, to collect and analyze the necessary data.
- It is difficult to find personnel with the appropriate skills to perform the empirical studies.

Still, if our goal is to be able to identify characteristics of fault-prone code, it is essential that we perform such large case studies.

With this in mind, we have begun analyzing data that are routinely collected for all production systems at AT&T. Whenever a fault is identified in a software system, any time during the development or production lifecycle, an entry is made in a fault-reporting database system associated with that particular project. Included with each entry is the stage during the lifecycle that the problem was first observed, the release of the program during which it was observed, the release of the program during which it was fixed, the severity of the problem, all files modified to fix the problem, and other associated information. Also included in the database is source code for every version of each file in the system.

The ultimate goal of our work is to help determine a way to identify particularly fault-prone files. If it could be determined early on which files are likely to contain the most faults, development and test teams would have guidelines on how to allocate scarce resources so as to optimize the payback from testing and debugging effort.

Although that is our goal, we do not expect to be able to determine this with a single case study that investigates one system, even though we have at our disposal comprehensive data from many successive releases. However, we do expect to provide some useful evidence that can contribute to the attainment of this goal, and also, help delineate appropriate questions to ask in future case studies. In this way we expect to eventually be able to identify these types of characteristics. If we see the same phenomena occurring for different software systems, produced in different environments, then this provides evidence that the results we observe are generalizable.

This paper is organized as follows: Section 2 describes related work. Section 3 gives a high-level description of the system. The results of the actual case study are presented in Section 4, while Section 5 describes our conclusions, and plans for future work.

2. RELATED WORK

Fenton and Ohlsson [3] presented results of an empirical study designed to examine some of the standard "wisdom" about the distribution of faults and failures in software systems. They studied two successive releases of a large commercial system. They found evidence that some of the commonly assumed hypotheses hold, but in several cases they found strong evidence that the opposite situation prevailed. For example, while the general wisdom was that modules that were found to be particularly buggy during pre-release testing contained the highest concentration of faults discovered once the software was released to beta sites or to the field, this was not the case for the modules they examined. The rationale for the prevailing wisdom is that certain modules are particularly or inherently difficult and therefore contain many faults throughout their development and operational lifecycles. In contrast, as predicted by the standard wisdom, they *did* find evidence of a Pareto-like distribution of faults and failures, with relatively few modules containing a large proportion of the faults, during both pre-release and post-release stages.

An earlier study by Adams [1] done at IBM, focused primarily on the usefulness of doing preventive servicing of software systems. His primary findings were that most defects occur very rarely in practice, and that a very small percentage (roughly 10%) of all faults detected correspond to the bulk of the defects "worth fixing" because they will lead to serious field outages. Of course, it is very difficult to determine a priori which will be the faults worth spending the effort to correct. In our environment, every fault has a severity assigned to it. In practice, this assessment provides guidelines for which faults get fixed in the nearterm, and which are postponed, perhaps indefinitely. Thus, a Severity 1 fault gets top priority, with the goal typically being that it be fixed within 24 hours. In contrast, a Severity 4 fault has very low consequence of failure, and therefore its correction may be put off until higher severity faults have been removed.

Several studies, including [2, 3, 5, 6], have investigated the relation between module size and both the fault count and fault density of a module. A key question is whether small, medium, or large modules are most fault-prone. They have generally found that, contrary to common belief, as the size of the modules increases, the number of faults per unit size actually decreases. These results call into question years of effort to make modularization and design decomposition standard practice. This argument has been based on intuition and plausibility, stating that by limiting module size, it is easier to comprehend what the module is doing, and therefore fewer faults will enter the software, thereby raising the software system's quality. But empirical studies described in the above-cited papers seem to indicate that not only is this *not* the case, in fact the opposite situation prevails.

Graves et al [4] studied a large telephone switching system, and found that two of the best predictors of faults in a module were the number of times that the module had been changed in the past, and *module age*, defined as a weighted average of the size of changes times age of changes (number of lines modified) x (elapsed time since changes were made). This study also found that a module's size and complexity were generally poor predictors of faults.

Fenton and Ohlsson carefully point out that the fact that they found either weak or strong evidence supporting or refuting a given hypothesis does not necessarily mean that the same results will be found for other software systems. Adams made a similar comment. Fenton and Ohlsson also noted that there are not necessarily causal relationships between hypotheses and conclusions that appear to follow from them.

In this paper we will investigate questions similar to those posed by Fenton and Ohlsson for a different large industrial software system produced in a different environment. The more evidence that can be collected that certain hypotheses are supported in different environments, the more likely it is that these results are generalizable. Of course, if opposite situations are observed for some questions, then we know that these conclusions cannot be considered to be general occurrences.

Our study does differ in some important ways from that of Fenton and Ohlsson. Whereas Fenton and Ohlsson had two successive releases of their system at their disposal, we have data collected from thirteen successive releases for the system upon which we based this study. The data collected by Fenton and Ohlsson were from four different lifecycle stages of the system, namely function test, system test, "the first 26 weeks at a number of site tests", and approximately the first year of operation after the site tests. They grouped these into two phases: pre-release (function and system test) and post-release (site tests and first year of operation). Our data is much more extensive, covering faults discovered from the requirements through general release; a total of nine distinct phases for each release. In order to be able to compare our results with Fenton and Ohlsson, we will at times also group our results into *pre-release* (including development and unit, integration, and system testing), and *post-release* (including the beta release, controlled release, and general release phases). We will also sometimes find it useful to group development and unit testing together as *early-pre-release*, and group integration and system testing together as *late-pre-release*.

Another difference between this study and the one done by Fenton and Ohlsson is that theirs does not differentiate between the severity of faults. The data we have collected associates a severity with every fault, ranging from the most severe, Severity 1, which requires the problem to be fixed very quickly in order for the system to continue operating, down to the least severe, Severity 4, which are largely cosmetic or very minor faults. In addition to examining some of the questions that Fenton and Ohlsson did for their system, we will also examine the extent to which various issues hold regardless of severity and those which are different for different severity faults.

			Early-Pre-Release		Late-Pre-Release		Post-Release		
Release	Files	KLOC	Dev	Unit	Int	Sys	Limited	Gen'l	Total
1	584	146	7	763	2	218	0	0	990
2	567	154	2	171	3	24	0	1	201
3	706	191	15	387	0	85	0	0	487
4	743	203	0	293	0	31	0	4	328
5	804	232	2	282	13	30	2	11	340
6	867	254	1	287	5	33	0	13	339
7	993	292	14	156	7	12	1	17	207
8	1197	339	14	363	28	77	2	6	490
9	1321	377	50	298	28	50	2	8	436
10	1372	396	84	119	7	24	1	11	246
11	1607	427	17	158	17	71	4	14	281
12	1740	476	62	130	8	53	1	19	273
13	1772	471	35	52	1	26	2	9	125
Total			303	3459	119	734	15	113	4743

Table 1: Distribution of Faults

The referenced papers use the term *module* to describe the basic code components of their studies, referring either to collections of files, or to separate executable components. The basic components in our study are *files*, most of which are java files consisting of class definitions. Throughout this paper, when referring to results of the referenced studies, we be consistent with their terminology and use the term module. For results of our study, we use the term file.

3. SYSTEM DESCRIPTION

In this section we provide information about the system under test that served as the basis for this case study. Certain details have been omitted to protect proprietary information.

The system is an inventory tracking system. As mentioned above, we have data collected from thirteen successive releases of this product, produced over a period of several years. Fault data was collected during each of nine periods: requirements, design, development, unit testing, integration testing, system testing, beta release, controlled release, and general release, and was recorded in a database associated with the product. Since very few faults were identified during the requirements and design phases, we do not include that data in this study. We have also combined the faults detected during beta release and controlled release into a single category called 'limited release'.

The current version of the system contains 1,974 separate files, with a total of roughly 500,000 lines of code. Most of the system is written in java (1,412 files), with additional files written in shell scripts (105 files), Makefiles (102 files), xml (98 files), html (66 files), perl (35 files), c (19 files), sql (15 files), awk (8 files), and several other specialized languages. We removed from consideration non-code files like MS Word files, gif, jpg, or readme files. They are not included in the file count or code size mentioned above, nor will they be included in the actual study.

Java files range in size from 13 lines of code to more than 8,000 lines of code, with only 20% of all files larger than 300 lines. Of course, each release contains different numbers of files and therefore different amounts of code. This data is tabulated in Table 1. In most cases, as the system matured and more functionality was added and faults were fixed, it grew both in terms of the number of files and the number of lines of code. At the same time, the overall number of faults detected during succeeding releases generally declined.

During all thirteen releases, and all stages of development, a total of 4,743 faults were detected, the vast majority during testing, before the system was released to users. Table 1 shows the distribution of faults by release and by stage discovered. Of the 4,743 faults, only 78 (1.6%) were classified as being Severity 1, the most critical category of fault, and in no case did the Severity 1 faults exceed 5% of the faults identified during a release. Severity 1 faults require immediate correction in order for the system to continue to operate properly, and hence they are the ones of greatest concern to the organization. It is therefore particularly important to note that of the 78 Severity 1 faults, a total of only ten occurred once the system had been released to the field, spread across the thirteen different releases. 687 faults (14.5%) were classified as Severity 2, and 131 (2.8%) were considered to be Severity 4. Over 81% of the faults, 3,847, were categorized as Severity 3. We see that the vast majority of the faults across all releases (more than 97%) were detected prior to beta release. This is testimony to the effectiveness of the testing and assessment process for this system.

Following the convention used by Moller and Paulish [6], if the cause of a failure was corrected by modifying n files, we counted it as n distinct faults. This assures that every fault is associated with a unique file. Fenton and Ohlsson [3] also state that every fault is associated with a single file.

4. THE STUDY

We now describe each of the four categories of questions we addressed for this system, along with our findings.

The primary set of questions involves examining how faults are distributed over the different files, and to what extent faults are heavily concentrated in a relatively few files. We will consider the release during which a fault was discovered, the lifecycle stage at which it was first detected, and its severity. Additionally, we will analyze the distribution of

faults between newly added files and those that appeared in earlier releases. These results are described in Section 4.1.

The second category of questions involves the density of faults among different files. This includes looking at questions such as whether large files are more fault-prone than small ones in the sense that they have higher numbers of faults per 1,000 lines of code than do smaller files, and whether files with high fault densities at early stages of the lifecycle, also have high fault densities during later stages. Our observations in this area are described in Section 4.2.

The third type of question involves investigating the persistence of faults by comparing the fault distribution found during pre-release to that of faults uncovered during post-release, and from one release to another. In particular, we would like to see whether or not those files that have particularly high numbers of pre-release faults, also have high numbers of post-release faults. As mentioned above, the conventional wisdom relating to this states that some files are inherently complex and therefore will contain large numbers of faults throughout their development and production lifecycle. Fenton and Ohlsson [3] found the opposite to hold for the system they studied, and offered a plausible explanation for why that might be the case. In Section 4.3 we will see whether the data for this system support or contradict such a hypothesis. We will also investigate whether files with particularly high numbers or densities of faults remain buggy from release to release.

Finally, we consider whether files that appear for the first time in a given release are likely to contain more faults than files that have appeared in earlier releases. We explore questions relating to this distinction in Section 4.4.

4.1 Pareto Distribution of Faults

We began our study by investigating whether or not there was evidence of a Pareto-like distribution of faults. The prevailing wisdom says that we can expect that a small percentage of the files will contain a large percentage of the faults detected at every stage of the lifecycle. We first considered this issue for faults of any severity, and any stage of development. We then investigated whether the distribution was different for pre-release faults than it was for post-release faults. Finally, we particularized it for each of the four severity categories that can be assigned to a fault. In this way we should be able to see whether or not the distribution changes significantly for different severities, and also whether the distributions are different as the system matures.

Fault Concentration By Release

For each release, the faults were always heavily concentrated in a relatively small number of files. Table 2 summarizes this distribution data for each of the thirteen releases, both from the point of view of how they are concentrated in files, and the concentration based on the size of the files. Although there are many different ways that one can measure size, in this study we use lines of code as the measure. We also use the term 'code mass' when we are speaking of this size.

For Release 1, Table 2 indicates that 10% of the files account for 68% of the faults and 35% of the code, and 100% of the faults are contained in 40% of the files, accounting for 72% of the code mass. For Release 10 through Release 13, the entry in column 2 listing the percentage of faults found in 10% of the files is shown as 100%. In fact, as can be seen in the corresponding fourth column, 100% of the faults for those releases were concentrated in less than 10% of the files. We have used an entry of '-' for those releases in the third column rather than enter a misleading number. The actual percentages of the lines of code contained in these files are shown in the last column of the table.

The numbers presented in this table provide strong evidence that a small number of files contain most of the faults discovered during the entire lifecycle for every release, with the concentration getting stronger in later releases, as the product matures. For example, all the faults uncovered during Release 1 were contained in 40% of the files, while in Release 13, they were contained in just 4% of the files.

Our next goal was to see whether this skewed distribution could be explained by the fact that the files containing most of the faults also contained most of the code. The columns labeled '% LOC' in Table 2 show this information for each release for both the 10% of the files containing the largest number of faults and for the set of files that contain 100% of the faults in that release. For Release 2, for example, we found that the 10% of the files that accounted for the highest number of faults contained 33% of the lines of code, while the 16% of the files that contained the entire set of faults contained a total of 36% of the code for this release. For all of the releases, the percentage of the code mass contained in the files containing faults exceeded the percentage of the files. This partly explains the skewed distributions of faults when viewed solely in terms of the number of files irrespective of their sizes, but it certainly does not explain the phenomenon entirely.

As before, we saw increasingly strong evidence as the product matured through later releases that a relatively small percentage of lines of code accounted for most of the faults. All of the files that contained faults found during Release 1 accounted for 72% of the lines of code, while for Release 13, that number dropped to 13%. We therefore see evidence that it would be worthwhile to concentrate fault detection and correction effort in the relatively small number of fault-prone files if they could be identified early.

Figure 1 shows graphically how the distribution of faults changed as the system matured, becoming successively more concentrated in fewer files. We included the information for five releases so that it would be easy to see this progression.

The cumulative percentage of faults and code size are shown graphically for Release 12 in Figure 2. We see that roughly 7% of the files contain 100% of the faults and 24% of the lines of code. All of the other releases have similarly shaped curves, with the percent of faults in the files growing much more quickly than the percent of code mass. In addition, the percent of the system's code mass included in the faulty files always exceeded the percent of the files that contained the faults.

As mentioned above, the summary data for the thirteen re-

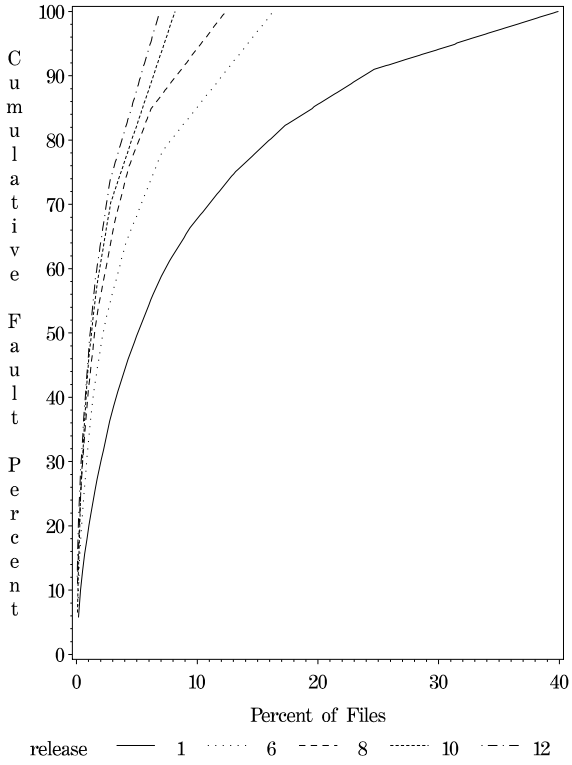

Figure 1: Fault Distribution for Releases 1, 6, 8, 10, 12

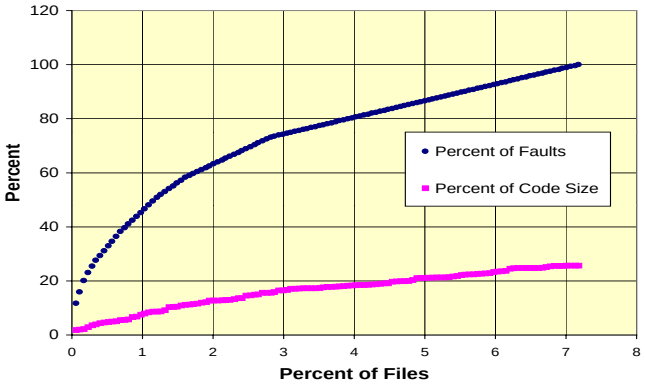

Figure 2: Cumulative Percentage of Faults and Code Size - Release 12

	10% Files Contain		100% Faults Contained In	
Release	% Faults	% LOC	% Files	% LOC
1	68	35	40	72
2	85	33	16	36
3	83	33	20	43
4	88	37	15	42
5	85	41	16	46
6	92	33	13	34
7	97	32	11	32
8	94	32	12	38
9	96	30	11	31
10	100	-	8	26
11	100	-	7	26
12	100	-	7	24
13	100	-	4	13

Table 2: Overall Pareto Distribution By Release

	Early-Pre-Release		Late-Pre-Release		Post-Release	
Release	% Files	% LOC	% Files	% LOC	% Files	% LOC
1	36	67	18	43	0	0
2	15	35	3	11	0	3
3	18	41	7	22	0	0
4	14	39	3	12	1	4
5	12	39	4	17	1	4
6	12	32	3	16	2	7
7	9	28	2	6	1	6
8	11	34	4	19	1	1
9	9	27	4	12	1	5
10	7	22	2	9	1	3
11	5	18	3	16	1	4
12	6	20	2	11	1	5
13	3	10	1	7	1	4

Table 3: Distribution of Faults By Lifecycle Stage

leases shown in Table 2 provides strong evidence that there is a very uneven distribution of faults among files for every release, with the evidence becoming stronger as the product matures with succeeding releases. Of course, if the system had remained essentially the same, with changes to the system representing only fault corrections, this would not be surprising. However, an examination of the second and third columns of Table 1 show that the size of each successive release has generally increased significantly, demonstrating that in most cases, new functionality has been added at each successive release. In fact, it more than doubled, both in terms of number of files and lines of code, as the system evolved from Release 1 to Release 8, and almost tripled in size between Release 1 and Release 12. This significant growth in size makes this result far more exceptional than if the system remained essentially static in size from release to release.

Fault Concentration By Stage

After investigating the overall fault distribution for each of the releases, we wanted to see whether there were distinct differences in the fault distribution depending on the stage of development during which the fault was first observed. The data for each of the releases are summarized in Table 3.

We defined early-pre-release to include development and unit testing. For all releases, faults first detected during early-pre-release accounted for a substantial majority of all faults, in most cases over 80%. It is therefore not surprising that for each of the releases we considered, the fault distribution for this phase looked similar to the overall fault distribution.

Late-pre-release was defined to include integration and system testing. As can be seen in Table 1, there were many fewer faults detected during this phase than during early-pre-release for every release, and so the faults were concentrated in a smaller percentage of the number of files, and the code mass contained in those files was also significantly smaller.

We considered post-release faults to include any faults that occurred either during limited release or during general release. Again by looking at Table 1 we see that there are generally very few post-release faults detected, and hence these faults are generally concentrated in less than 1% of the files, accounting for just a few percent of the code mass for each release.

In summary, we found strong evidence that a small percentage of files accounted for most of the faults, for all of the

releases, for faults first detected at each of these stages.

Fault Concentration By Severity

Together, Severity 1 faults and Severity 4 faults accounted for only 4% of the faults uncovered during the thirteen releases. The vast majority were Severity 3 faults (81% overall) with the remaining 15% being Severity 2 faults. These overall percentages were typical of those found during individual releases.

The very small numbers of Severity 1 and Severity 4 faults guaranteed that they would occur in each release in a very small percentage of the files. For Severity 1 faults, this ranged from 3% of the files in Release 1 to none in Release 12. For Severity 4 faults the range was from 4% of the files in Release 1 to 0.3% of the files in Release 12.

Even though there were a larger number of Severity 2 faults than Severity 1 and Severity 4 faults, they still represented a relatively small number of faults for each release and thus also always occurred in only a small percentage of the files.

Since Severity 3 faults accounted for roughly 80% of the faults in most releases, it was not surprising that their concentration in files was typically very similar to that of all faults. For example, for Release 1 all Severity 3 faults occurred in 36% of the files, accounting for 68% of the lines of code. This compares to 40% of the files and 72% of the lines of code for all faults. For Release 8, the numbers were 11% of the files and 35% of the lines of code, as compared to 12% and 38% respectively. Release 12 had all of its Severity 3 faults concentrated in just 6% of its files, accounting for 22% of the lines of code. All the release's faults were in 7% of the files and 13% of the code mass.

The fact that we have seen strong evidence of a Pareto principle whether we analyzed the system by release alone, by release and phase of development, or by release and fault severity, indicates that this is certainly something worth considering when deciding how to focus testing effort. This is especially true since this phenomenon has been reported by other researchers who have studied different systems, developed in different environments, using different programming languages [1, 3, 7].

4.2 Effects of Module Size on Fault-Proneness

Another issue we considered was how the size of code components affected the number and density of faults. It has been argued for a long time that large modules are much more fault-prone than small ones. For many years, therefore, it has been considered good programming practice to keep modules small. In fact, many organizations have formalized that idea and provided strict guidelines, often requiring that all modules be kept smaller than some specific size.

A few earlier empirical studies, including those described in [2] and [6], found that, contrary to this standard wisdom, there was no evidence that larger modules had higher fault densities than smaller ones, and that, in fact, just the opposite was true. The data in our study generally supports these results. Figure 3 plots fault density in faults per 1,000 lines of code against the size of the files, for all files containing faults, for Release 12. We created the same plots for each of the other releases; in all cases, the shapes of the plots were similar. Figure 3 shows fault densities between 10 and 75 faults/KLOC for the smallest files (under 100 lines), and leveling off at 2-3 faults/KLOC for files larger than 1000 lines.

Hatton [5] compiled the results of several studies, and observed that fault density was high for the smallest components, decreased to a minimum for "medium-size" components, and then started increasing again as component size grew. The minimum fault densities appeared in components with approximately 200-400 statements, both for assembly code and Ada programs. In contrast to this observation, the fault densities for the files of our system do not increase as the files get larger.

Fenton and Ohlsson's data, however, agree neither with our results nor with those in [2, 5, 6]. In examining the relation of fault density to file size, they found no trend at all. Figure 9 in [3], showing no apparent relation between file size and fault density, is the the exact analog of our Figure 3, which shows a clear inverse relationship.

Various other factors, including whether or not a file is new, the amount of changed code, the amount of testing performed, the amount of operational use that a file experiences, and the experience of the programmer, might be responsible for the fault density of a particular file. The systems examined in the studies mentioned covered a wide range of development environments, programming languages, and applications. Although a variety of reasons have been proposed for this observed phenomenon, no definitive explanation has yet been proven. At this stage in our study, we are unable to offer an explanation; we hope to investigate the issue further in future studies.

Since the number of pre-release faults completely dominates the number of post-release faults for each of our releases, it is not surprising that we observed nearly identical plots when we restricted the domain to only pre-release faults. There were too few post-release faults to say anything meaningful about their distribution relative to file size.

4.3 Persistence of Faults

In this section we consider whether or not those files with high concentrations of faults detected during pre-release also tend to have high concentrations of faults detected during post-release. We also consider whether faultiness persists between releases, i.e., whether files with relatively high numbers of faults in early releases are likely to have high numbers of faults in later releases. The implication of either of these hypotheses being true is that it would help us identify files that are likely to be unusually fault-prone, and therefore help us determine particularly good ways of allocating testing resources.

Fenton and Ohlsson [3] compared the number of faults per module found in pre-release testing of a release against the number found in post-release use of that release. Their study involved two successive releases. They found that many of the post-release faults occurred in modules that contained no pre-release faults, and 100% of the post-release faults

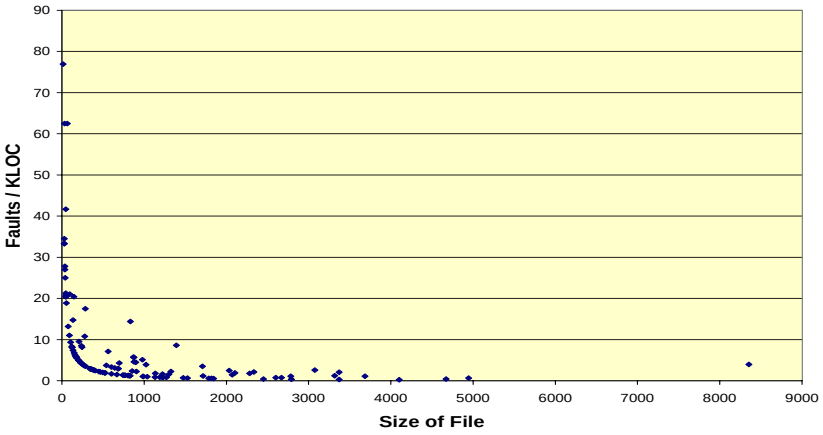

Figure 3: Fault Density vs. File Size - Release 12

Release	No Late-Pre-Release Faults	At Least 1 Late-Pre-Release Fault
1	0	0
2	1	0
3	0	0
4	3	1
5	9	4
6	8	5
7	18	0
8	5	3
9	5	5
10	7	5
11	10	8
12	14	6
13	7	4

Table 4: Distribution of Post-Release Faults

occurred in modules that had either 7% (in one release) or 23% (in the other release) of the pre-release faults. They interpreted these results as being in direct contrast to the common belief of persistence of faultiness in files.

We did a similar evaluation of the late-pre-release and post-release faults for the 13 releases. Our results thus address the question of whether a higher incidence of faults in integration and system testing implies a higher incidence of faults in post-release operation. We found results very similar to Fenton and Ohlsson's across all thirteen of the releases we studied. The percent of late-pre-release faults that occur in files with *no* post-release faults ranges from 72% through 94%; correspondingly, 100% of the post-release faults occur in files that account for 6% through 28% of the faults discovered during late-pre-release.

However, Table 1 shows that no release contains more than twenty post-release faults, and roughly half of them contain ten or fewer post-release faults. Considering that each release contains between 584 and 1,772 files, we are forced to conclude that there is simply not enough data to draw any meaningful conclusions about the predictive ability of fault concentration during pre-release for post-release from this analysis.

There is, however, an indication that the files that contain pre-release faults are not the most-likely place where post-release faults will occur. Table 4 shows, for each release, how the post-release faults were distributed. Column 2 indicates the number of post-release faults found in files that contained no late-pre-release faults, while column 3 lists those that were found in files containing at least one pre-release fault. For every release, at least as many post-release faults were found in files that had *no* late-pre-release faults, as were found in files that did contain some late-pre-release faults. In fact in Release 7, all eighteen post-release faults occur in files with no pre-release faults. Again, the relatively small number of post-release faults in any release prevents us from drawing any firm conclusions from these numbers, but we intend to examine the question further in future studies of additional software systems.

Release	1	2	3	4	5	6	7	8	9	10	11	12
Rel (n-1)	-	27	54	21	45	42	52	34	21	17	39	24
Rel (n+1)	63	46	27	30	56	34	34	27	22	36	22	-

Table 5: Persistence of High-Fault Files

	% Faulty Files		Faults/KLOC	
Release	OLD	NEW	OLD	NEW
2	15.4	16.3	1.29	1.46
3	18.5	24.8	2.32	4.01
4	13.8	39.1	1.42	5.44
5	13.8	31.0	1.33	2.09
6	11.0	36.8	1.13	4.90
7	10.2	13.3	.69	.90
8	12.2	12.8	1.36	1.97
9	8.9	36.3	.81	4.85
10	7.6	20.7	.60	1.15
11	6.9	7.9	.60	1.54
12	5.8	14.4	.49	1.08

Table 6: Comparison of Faultiness For Old and New Files

To assess the persistence of faults between releases, we traced individual files through successive releases of the system. For this purpose, we defined 'high-fault files' of a release to be the top 20% of files ordered by decreasing number of faults, plus all other files that have as many faults as the least number among the top 20%. We then did pair-wise intersections of the sets of high-fault files between releases, generating for each pair of releases, the set of files that are high-fault in both releases. This allowed us to see which high-fault files of each release had already been high-fault in any previous releases, and which ones continued to be high-fault files in later releases. Although we determined these values for all pairs of releases, Table 5 only shows this backward and forward fault persistence for a release's predecessor and successor. Row 1 shows the percent of high-fault files in Release (n-1) that remained high-fault files in Release n, while row 2 shows the percent of high-fault files in Release (n+1) that had been high-fault files in Release n.

The numbers provide moderate evidence that files containing high numbers of faults in one release, remain high-fault files in later releases. 17% to 54% of the high-fault files of Release n are still high-fault in Release (n+1), with the persistence being greater than one-third in more than half of the cases. The percent of high-fault files in Release n that were also high-fault in Release n-1 ranges from 22% to 63%. Release 12 was particularly interesting. A sizable percent of the high-fault files during this release were also high-fault in every one of the previous releases; more than 40% of its high-fault files were also high-fault in Release 1, which was produced several years earlier.

4.4 Old Files, New Files

In this section we explore the fault-proneness of files based on whether they are newly written in the current release, or pre-existed in an earlier release. For each release beyond the first, we computed the percentage of pre-existing files for which faults were identified, and also the percentage of new files that were shown to contain faults. These values are shown in columns 2 and 3 of Table 6, respectively. In every release, the percentage of faulty new files is larger than the percentage of faulty pre-existing files.

Table 6 also shows the fault density for pre-existing files and for new files. It is not surprising that for every release, the fault density is higher for new files than for pre-existing ones.

Although these observations simply confirm our intuition, they nonetheless provide valuable insights. In particular, given that this was always the case for the system used in this case study, it might be reasonable to suggest that test teams allocate more resources for testing new files than for testing pre-existing ones, since both larger percentages of new files have been shown to contain faults, and their fault density is higher as well.

5. CONCLUSIONS

We have considered a series of thirteen releases of a large, evolving industrial software system as the subject of a case study intended to evaluate issues related to fault distribution and fault-proneness of files. We investigated whether faults concentrate in small numbers of files and small percentages of the code mass, and found that they did. Not only did we find this to be true for every release, we found that as the product evolved, the faults became increasingly concentrated in increasingly smaller proportions of the code. In particular, in the first release, all of the faults were contained in 40% of the files that contained 72% of the lines of code of the system. By the last release, all of the faults were concentrated in just 4% of the files, containing 13% of the code mass.

We also investigated the concentration of faults by stage of the lifecycle. We considered faults discovered during development, three stages of testing, and three levels of customer release. We grouped those faults into early-pre-release, late-pre-release, and post-release faults, and compared the distributions. We found very few post-release faults during any of the thirteen releases, with twenty being the largest

number of such faults identified during any single release. For that reason, post-release faults were very highly concentrated. In each of the releases, they appeared in at most 2% of the files, accounting for no more than 7% of the lines of code. For each release, the early-pre-release faults accounted for a clear majority of the faults, in most cases more than 80%. For that reason, the distribution and concentration of early-pre-release faults looked very much like the overall distribution, with a definite trend showing an increasing concentration of early-pre-release faults as the product matured from Release 1 to Release 13. There were also relatively few late-pre-release faults uncovered during any release, and so they were also very heavily concentrated. With the exception of two early releases, all late-pre-release faults always appeared in under 5% of the files.

In all cases, the percentage of lines of code contained in files that contained faults exceeded the percentage of files that contained faults, thereby partially explaining the Pareto-like distribution of faults. However, in no case did this account entirely for the extreme nature of the fault concentration.

We examined the persistence of faultiness of individual files, both within a single release, and across successive releases. Although our data do not give strong results for either case, they indicate that faults are likely to be found both in previously faulty files and in files that were previously fault-free. Within a single release, we tried to determine whether the number of pre-release faults can be used to predict the number of post-release faults attributed to a file, and decided that the very small total number of post-release faults in our data prevented drawing any conclusions of this sort. We did note, however, that in every release half or more of post-release faults occurred in files that had no late-pre-release faults. This observation contradicts the conventional wisdom that the best places to find bugs are places where they have already been found.

Across successive releases, we found some evidence that high-fault files of one release tend to remain high fault in later releases. In particular, between a quarter and two-thirds of the high-fault files of Release n were high-fault in the previous release, and up to one-half continued being high-fault in the following release. This observation indicates that it is a mistake to shortchange testing efforts for previously high-fault files, under the belief that their problems have all been solved in the previous release.

Acknowledgments

This work could not have been performed without the cooperation and generous assistance of various members of both development and testing teams at AT&T. Danielle Bellavance graciously gave us access to both data and personnel. We are very grateful to Cheryl Bliss, Jainag Vallabhaneni, Jim Laur, and Joe Pisano for their time and insights. Beto Avritzer was very helpful in assisting us with the figures, and Parni Dasu provided suggestions on appropriate statistical analysis as well as help with the figure generation. We also appreciate the assistance we received from our colleagues Ken Church, David Korn, and John Linderman.

6. REFERENCES

[1] E.N. Adams. Optimizing Preventive Service of Software Products. *IBM J. Res. Develop.*, Vol28, No1, Jan 1984, pp.2-14.

[2] V.R. Basili and B.T. Perricone. Software Errors and Complexity: An Empirical Investigation. *Communications of the ACM*, Vol27, No1, Jan 1984, pp.42-52.

[3] N.E. Fenton and N. Ohlsson. Quantitative Analysis of Faults and Failures in a Complex Software System. *IEEE Trans. on Software Engineering*, Vol 26, No 8, Aug 2000, pp.797-814.

[4] T.L. Graves, A.F. Karr, J.S. Marron, and H. Siy. Predicting Fault Incidence Using Software Change History. *IEEE Trans. on Software Engineering*, Vol 26, No 7, Jul 2000, pp.653-661.

[5] L. Hatton. Reexamining the Fault Density - Component Size Connection. *IEEE Software*, March/April 1997, pp.89-97.

[6] K-H. Moller and D.J. Paulish. An Empirical Investigation of Software Fault Distribution. *Proc. IEEE First Internation Software Metrics Symposium*, Baltimore, Md., May 21-22, 1993, pp.82-90.

[7] J.C. Munson and T.M. Khoshgoftaar. The Detection of Fault-Prone Programs. *IEEE Trans. on Software Engineering*, Vol18, No5, May 1992, pp.423-433.