Component Testing Is Not Enough - A Study of Software Faults in Telecom Middleware

Sigrid Eldh[1,2], Sasikumar Punnekkat[2], Hans Hansson[2], and Peter Jönsson[3]

[1] Ericsson AB
[2] Mälardalens University
[3] Combitech
Ericsson AB, Kistagången 26, Stockholm, Sweden
sigrid.eldh@ericsson.com

Abstract. The interrelationship between software faults and failures is quite intricate and obtaining a meaningful characterization of it would definitely help the testing community in deciding on efficient and effective test strategies. Towards this objective, we have investigated and classified failures observed in a large complex telecommunication industry middleware system during 2003-2006. In this paper, we describe the process used in our study for tracking faults from failures along with the details of failure data. We present the distribution and frequency of the failures along with some interesting findings unravelled while analyzing the origins of these failures. Firstly, though "simple" faults happen, together they account for only less than 10%. The majority of faults come from either missing code or path, or superfluous code, which are all faults that manifest themselves for the first time at integration/system level; not at component level. These faults are more frequent in the early versions of the software, and could very well be attributed to the difficulties in comprehending and specifying the context (and adjacent code) and its dependencies well enough, in a large complex system with time to market pressures. This exposes the limitations of component testing in such complex systems and underlines the need for allocating more resources for higher level integration and system testing.

Keywords: Software, Fault Classification, Fault distribution, Testing.

1 Introduction

We have investigated a number of failures in a part of a subsystem of a large complex middleware system at Ericsson. Based on this investigation, this paper identifies classes of failures, presents their frequencies, and isolates the failure classes that are caused by software faults. Our overall motivation and planned future work is to create a controlled experiment by re-injecting known faults of different types in the code, to be able to learn more about efficiency and effectiveness of various test techniques [1]. We initially planned to use published information on commonly prevalent software faults, fault classes and their frequencies, but were unable to find sufficient information from existing literature. Since failures are related to software faults in

A. Petrenko et al. (Eds.): TestCom/FATES 2007, LNCS 4581, pp. 74–89, 2007.

complex and intricate manners, we believe that a realistic characterization of their correlation will be helpful in determining effective as well as cost-efficient testing strategies. Our objective is to understand how faults and failures manifest themselves, especially in the context of 'real' faults that occurred in commercial or industrial systems.

Most software industries do not pay enough attention to understand the typical faults that exists in their software. These industries collect every anomaly and complaint, from both verification teams and customers, but seldom faults and failures found by designers. Hence, failures are collected from the later stages in the software process, saved in databases, classified based on priority, status of management (e.g. analyzed, fixed, tested, approved) and classified based on organization or software sub-system where the actual fault is believed to exist. Occasionally deeper analysis and classifications are done and root cause analysis (RCA) are performed, especially when major incidents involving customer complaints occur. Most classifications [8, 9, 30, 35] end prematurely by defining the failure on too high level to understand what software fault caused the failure. We suspect that this is the case in most industry and commercially available software, with the exception of safety critical software..

In this study, we consider the software in a typical telecom system. We have not looked at all the reported failures for the entire node, but focused on one particular part of the software, which we consider to be a typical representative of telecom middleware. From 4769 reported failures, we have selected a sample of 362 failures, which can be considered to be important since they were all corrected. This data has been collected during a period of three years based on 65 different designers and software integrators across the world. We chose these failures since their labels in the configuration management system made it easier to locate the corresponding software fault, compared to repeating the tedious troubleshooting and debugging of the system, which would otherwise have been required. The questions we primarily try to address with this case study are:

- What are the real and important faults in software that have propagated to failures, and subsequently fixed?
- What is the distribution of the failures and faults into various classes? (This classification will allow us to re-inject faults of the same type in the software, thereby providing a basis for our planned evaluation of test techniques.)
- Is there any other specific pattern of faults and failures that would guide us into understanding the software process better?

Our focus is on software faults, but our study has shown that just less than half of the reported failures are not a direct consequence of a software fault that can possibly be re-injected in the code. Instead, failures relate to a variety of problems, e.g. hardware, third party products, process issues, organization, and management issues. We decided to keep all information to give a better perspective for researchers trying to understand problems in the software industry, and better explain them as a part of our case study.

Outline: We discuss and define the terminology used in the next section and then related work in section 2. Section 3 describes the set-up of our case study and data selection. In section 4, we discuss our findings of the failure distribution and the different classes. Validation is discussed in section 5 and future work in section 6. We will end section 7 with the conclusions that we have derived from this case study.

1.1 Terminology

The related terminology in this area (fault, error, cause or reason, failure, bug, defect, and anomaly) is often confusing because these terms are used interchangeably and inconsistently by many in industry and academia; see further discussion in Mohaghegi et al [27]. Therefore we define the following terms with inspiration from earlier work from Avižienis & Laprie [10] and Thane [9], where a *fault* is the static origin in the code, that during dynamic execution propagates (in Figure 1 described as by a solid arrow) to an *error* (which is an intermediate infection of the code). If the error propagates into output and becomes visible during execution, it has caused a failure. An error or failure can both cause another fault to occur. At Ericsson, failures are reported as Trouble Reports (TRs). Occasionally, these TRs gives in their analysis section a more direct explanation of the cause of the failure, but mostly they only describe the symptoms. TRs are not uniquely identifying failures (i.e., several TRs may identify the same failure) and there is not a one to one relationship between a fault and a failure (i.e. different faults may lead to the same failure and some faults may cause multiple failures).

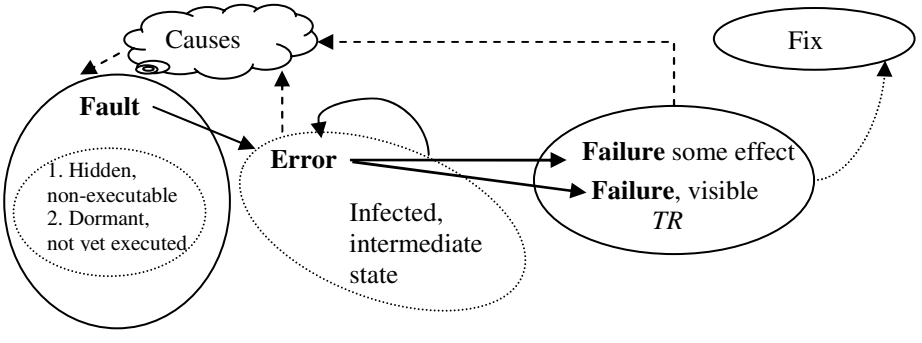

Fig. 1. Terminology mapping

A failure can in turn propagate to another part of the software and be the cause of another fault. One difference compared with Avižienis & Laprie is that we separate the actual cause of the fault from what manifests itself in the software. As an example, a faulty specification can lead to a fault in the software, but to find what code to re-inject in order to represent such a fault is not clear. It might be that the fault specification misses to define a case not implemented in the code, which leads to faulty assumptions and not adding extra paths when needed. We occasionally refer to the term “real fault”, meaning, a fault found in a commercial or industrial system.

2 Related Work

Our purpose is to investigate software test techniques is described in our position paper [1]. We noticed that most test technique investigations used small code

samples, often with very few faults injected [2, 4]. The faults used as the basis in test technique investigations are often invented and "simple" or made to prove a specific point [20]. This did not match our experience with faults in software for complex systems. Even if there exist attempts to create better faulty programs to use for test techniques research [3, 32, 33, 34], they still do not contain enough data from industry, and are relatively small compared to our complex middleware. Therefore, we argue that the early stages fault investigations for test technique research [5, 6, 7] needs to be updated. Andrews et al. [32], share a similar goal, but uses a different approach. They compared mutant generated faults with hand-seeded real faults, and concluded that the faults were different in nature, but shows statistical promise. Analyzing our result in contrast with theirs, it becomes evident that the type, nature of the fault and fault class, and its distribution *is not sufficiently explored* to draw strong conclusions about test techniques. We tried to find examples of classes and types of code faults to re-inject, but did not find any good list to use, instead we found several papers investigating real faults (and failures) classified for different purposes [8, 9, 11, 12, 13, 14, 15, 16, 18, 20, 23, 24, 35] and not distinguishing faults from failure or cause. In particular, for the more commonly used Orthogonal Defect Classification (ODC) [8], we concluded that the classes are classifying failures, and not faults, e.g. an interface failure (which is an ODC class) could be caused by several software code faults. Thus, these classifications are insufficient to support us in our aim. DeMillo and Marthur [13] have made an attempt to classify real software defects by automatic means. We have used these fault and failure classifications as inspiration to our classification and we will discuss them in the section on future work. Rather than adopting, we strongly propose that different software domains have different sets of fault and fault distributions, depending on organizations, languages, development and testing methods, as well as the ways of measurement. Conclusions on test techniques should be based on first creating a thorough fault analysis particular to the domain, but with known methods. This will provide better understanding of which typical failures and related faults that are relevant for this particular software. We realize that the gap between research and industry is wide [31], but we hope to close this gap by doing controlled research on commercial software in an isolated environment.

Huffman and Rothermel discuss in [29] the semantics of a fault. This is interesting research, since it implies that faults have a variety of impact, depending on the fault. Our research shows that some types of faults that affect the software are a combination of faults, and are definitely involving more than one file, and more than one entry in a file. There is a danger in inserting only single semantic faults even if they are dominating. Single semantic fault injection is the predominant way of injecting faults (and mutations). One fault can propagate into many different symptoms (which is one of the explanations to the high number of duplications). One fault might propagate and behave differently, depending on how "complicated" it is. Hyonsook and Elbaum [3] have with the work on SIR framework, used files from industry (SPACE and Siemens program [33, 34]), but also let experienced designers deliberately insert faults (also used by [32]). There is no way of telling if these faults are representative of common faults in any system, or if they are too simplistic in nature. Our initial reaction when analyzing failures have been that faults that

dominate are much more complicated in nature than plain logical or computational faults, which do occur, but not as frequent. Ishoda [21], argues that basing research by capture- recapture (inserting faults, and then estimating how many of them are found) is not a sufficient technique for reliability (*and test evaluations*) analysis, and that correct frequency and type of faults must be known for the software in question. We support this argument, which also lead us to investigate our own frequency and type, to understand the characteristics of our particular type of software. Ostrand, Weuyker, and Bell [22] work in the same domain as us, large middleware systems with similar problems. They have not focused on the fault type in itself, but on occurrences and location, which is similar to our approach. Since they have classified based on their MR (similar to Ericsson's TR), we also assume (but have not verified) that their data suffer from the same problem as our, what is reported (failure) and the connection to the actual fault in some code file needs a much further analysis. Yet, they report interesting results that seem to match our experiences: Most of the faults reside in (or are reported on) 20% of the software, i.e. 80% of the software is more or less fault-free. Furthermore, their distribution seems to match ours, with over 50% of the failures related to missing or spurious code. A comparison with our figures shows that their distribution and frequency is very similar to ours, even if it is done more than 20 years ago. We will discuss this further in our last section: Dicussions and conclusions.

The key problem we would like to address is that there is no recent industry data available for research purposes. In addition, people who measure are often using too high level classifications [8] to be useful for our purpose. We have also studied bug taxonomies [12, 32], but conclude that they mix cause, fault and failure, and are seldom providing obvious support for re-injecting faults, even if they give valuable information about failures.

Our main conclusion is that it is important to regularly collect and report findings of this nature from real industrial and commercially used systems to keep information in tune with development approaches, software and faults. We also assume that there is a large diversity of the frequency of faults, depending on what type of system and domain they reside in. We also suggest that within a domain – or type of system (e.g. a large complex operating system middleware with partially proprietary hardware) it is possible to find similar structures across the entire domain, which could indicate that the results are not limited to only e.g. telecommunication middleware systems.

3 Case Study Process and Data Selection

The process followed in our case study is described in Fig. 2. We started by selecting the Trouble Reports (TRs) for which a link to the corresponding code exist; using a script that automatically linked the fault id into the corrected code as a comment. Then we compared the corrected code with the original version of the code, to identify areas in the code where the fault could reside. This is a non-trivial task, since enhancements and improvements to the code are mixed with corrections. We then classified each Trouble Report into one of the chosen failure classes.

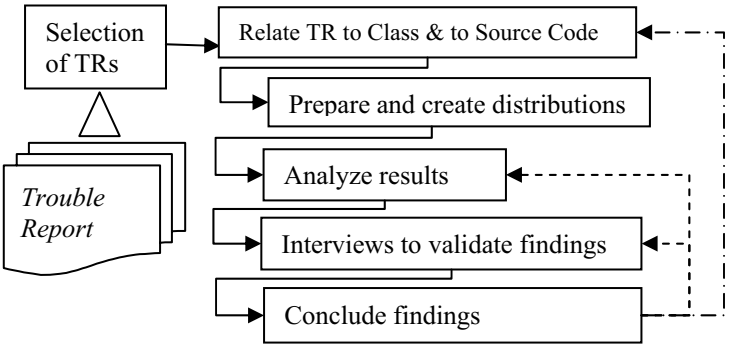

Fig. 2. Overview of the classification process

The distribution of failures was then analyzed, followed by some interviews of the designers, with the aim to validate our findings. The component size was approximately 180-200 000 lines of non-commented code (measured over the three year period). For the entire middleware system, there were 4769 TRs reported and of these we have considered 1191, indicating that it is a central part of the software. From these reported TRs (and also, from the complete set of 4769 TRs), some of the TRs have been analyzed to originate from faults elsewhere, or require corrections in two places. We must understand that not all of the 1191 TRs lead to a correction. There are 181 TRs reported directly on this component's code during this period. Our number is higher (362) which indicates that TRs that reside elsewhere (are reported on other places in the code) affects this code. The TRs within this target are all using a particular labeling function that makes it possible to trace the TR to the actual file (see Fig. 3).

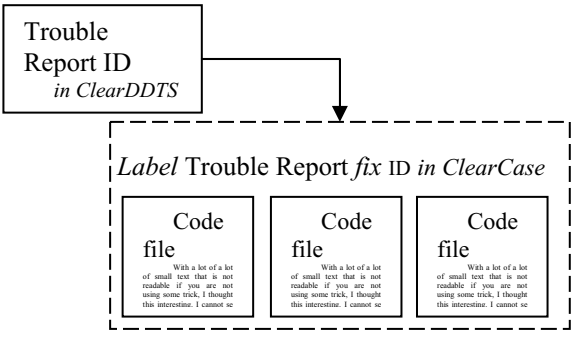

Fig. 3. Failure (Trouble Report) – fault/fix relation, where the label points out involved files

How this subset (of labeling) is related to the 1191 TRs we have not been able to pinpoint, since it would have required serious data mining, taking all change requests in to account and other factors. From the entire set of 362 TRs, we now withdraw all faults that are not software related, e.g. duplications and wrong usage, to come up

with our set of 295 unique failures, and of them only 170 (204 adjusted[1]) are software failures. This means that approximate 14% (17% adjusted) of the reported failures (1191 TRs) at least are real software code faults!

Little or no general tool-support exists to trace, categorize and get support in tracking down the fault causing a failure. Currently this has for non-trivial cases to be handled by manual work through code inspection. Manual comparisons of two versions of the source code is often the used method to extract the actual difference, but this is not enough, since enhances to the software and other modifications must be excluded and the fault origin or origins must be isolated. The problem is non-trivial in systems with a multitude of code branches, since it amounts to know where to look without having to scan through a multitude of files and to separate fault corrections from any other code modifications, improvements and change requests.

This is the true complication of relating a "unique" failure to a "unique" fault since they do not have a one to one relationship. What even more complicated our classification was that the original designers (and troubleshooters) were not available to ask of the real origin of the fault. Doing a complete trace could take us between 3 days to two weeks, which explains why this route to identify and gather faults from failures is seldom taken for this type of research. With the labeling function, the connection between the TR-system id and the actual place in the code where the fault could reside could be made, and capturing the correct code became a task of analysis between one hour and 2 days, when done by a person not familiar with the software. We consider this a great improvement.

4 Identified Failure Distributions

This section presents the fault classification, the distribution of failures observed in our case study together with the distribution of the software faults over number of affected code files.

4.1 Fault Classification

In our classification we use the following classes of faults

- *Language Pitfall* are faults that are specific to the programming languages used, e.g. pointer being null, pointer pointing to an invalid address, valid address but pointing at garbage. E.g. arrays is a common type of data structure accessed with index, and if the value falls outside the boundary of the array then the accessed element is unknown, or data might be modified that shouldn't. In addition, overflow and underflow are categorized in this class.
- *Computational/Logical faults* are faults inside a computational expression. A logical fault is similar to a computational fault, except that it is related to a logical expression.

[1] Our investigation showed that for many reported failures, the actual fault (represented by the corrections in the code) is not unique and failures and faults do not have a one to one relationship. Therefore, we have adjusted our figure, by adding all software faults that are contributing to the cause of a failure.

- *Fault of omission* is when a fault happens due to missing functionality, i.e. the code that is necessary is missing. E.g. a part, an entire statement or a block of statements missing can be classified as faults of omission, which means missing either of the following: function call, control flow path, computational or logical expression.
- *Spurious faults* are similar to Faults of omission, but in this case there is "too much code", and the correction is to remove one or several statements.
- *Function faults* are faults, such as calling the function with the wrong parameter or calling the wrong function.
- *Data faults* including faults of several types: Primitive data faults, which include defining a variable to the wrong primitive data type, e.g. integer to be unsigned but should not have been; Composite data fault e.g. structure (in C or C++); initialization fault and assignment fault.
- *Resource faults* are faults that deal with some kind of resource, such as memory or a time, therefore this class handles faults from allocation, de-allocation, race-conditions, time issues (dead-lock) and space (memory, stack etc).
- *Static code fault class is* code that does not change after compilation of software or after the first execution, when using an interpreted language. This code is e.g. source code or configuration files which the software when executing uses.
- *Third party faults* are faults in software for which we do not have access to the source code and hence cannot correct ourselves.
- *Hardware faults* and *Documentation faults* are self-explanatory classes, and do not relate to software.

We have grouped the above into four groups, viz., code, process, configuration management (CM) and other.

4.2 Fault Distribution

The selected of 362 trouble reports, were distributed as explained in Table 1 below, where faults of omissions topped the list. The second largest class of Trouble Reports is *Duplicates*. One failure out of three non-software related failures reported is a duplicate. What is interesting is that these duplicates were not identified as duplicates until they were corrected in the code. This means that the TRs were individually decided to be fixed, and assumed to be of different fault origin, since they showed different behavior and were viewed as different failures. Otherwise, these failures would have been omitted before ordering them to be fixed. The high number of duplication of failures is also related to the way testing is done on this complex system, and also to the type of software (which most other parts of the software were dependent upon). This means that many failures in this particular software will be found and reported by several persons. Another aspect is that failure reports are symptoms, and they might not appear in the same way (and be explained similarly) by two different persons. This problem is recently remedied, by enhancing the pre-test on the entire system before release (much as a result of the insight gained by this case study).

Table 1. Failure distribution into classes and their frequency. Second column is the adjusted value (when translating TR to fault classes).

Group	Fault/Failure Class	Failures	Code Fault adjusted
Code	Faults of omission	73	78
Process	Duplicate of TR	67	-
Code	Data fault	22	26
CM	No files associated	21	-
Process	Change requests	21	-
Code	Static code fault	20	21
Code	Spurious faults	17	17
CM	SCM	15	-
Code	Resource fault	13	13
Code	Computational/ Logical fault	11	28
Code	Function fault	11	13
Process	Fault not fixed	9	-
CM	Compile time fault	8	-
Other	Third party fault	5	-
Other	Documentation fault	4	-
Code	Language Pitfall	3	8
Other	Hardware fault	2	-
Process	Not a fault	1	-
Code/Other	Too difficult to classify	39	-
	Sum	**362**	**204**

Disregarding duplicates of failures leading to the same fault, we would focus on the 295 unique failures. However, the classification needs adjustment, in relation to how some failures are reported. For example, if a fault needs two corrections to be fixed, and there are duplicate TRs, there is no way of telling that one TR correction is assigned to one or the other fault correction, and the "duplicate" vice versa. We see this as a disturbance of the data, but have reported the adjustment for the fault classes in software. The adjusted values will if re-inserted cause failures. We have started further investigations to understand the nuances of how faults actually infect the software, and when and how visible they are.

The next interesting class of failures was "too difficult to classify" containing 39 of the failures. The reason is that it is impossible to pinpoint the exact difference, since the entire unit or file was re-designed, and large portions of the code rewritten. This also makes it difficult to pinpoint if the change was only due to the failure, or due to other factors such as updates, expansions, new features etc. We decided that for our purposes is not worth the effort to sort this out, instead we just conclude that 13% of the unique failures lead to a major change in the system. Only 8 (of 19) classes can be considered in our investigation, since these are the ones directly related to software faults. This is only 170 of 362 reported failures (47%), or 170 out of 295 unique failures (58%). This means that as much as 53% can be dismissed due to problems that are either indirect, process or system related, or disturbance in data and that these failures do not uniquely originate from the software. Third party faults are software faults, but they cannot be traced into code, since the source code is not always available to us, and must be corrected by another party. Another category is how the

system is built and integrated, including software configuration management (SCM) failures, compile time failures (during the build) and the category of "no files associated". These failures are strongly related to the fact that a major change of build system and product structure happened during the period of data collection, which explains why it constitutes 1 out of 5 non-software related faults.

In Table 2, we present distribution of faults together with the number of files which had to be updated to correct the faults.

Table 2. Distribution of adjusted faults into number of files

Code fault class	Faults	%	Number of files										
			1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**11**	**25**
Faults of omission	78	38.2	50	12	8	4	2	1		1			
Computational/ Logical fault	28	13.7	24	2		2							
Data fault	26	12.7	20	3	1		2						
Static Code Fault	21	10.3	4	7	2	1	1	3	1	1			1
Spurious faults	17	8.3	11	2	1	1	1					1	
Resource fault	13	6.4	8	5									
Function fault	13	6.4	11					1			1		
Language Pitfall	8	3.9	6	1	1								
Summation	**204**	**99.9**	**134**	**32**	**13**	**8**	**6**	**5**	**1**	**2**	**1**	**1**	**1**

The failures found in the case study, can involve between 1 to 64 files. The software faults can be distributed between 1 to 25 files. Analyzing this data shows the majority (66%) are from 1 file, but e.g. that faults of omission involve more than one file for 36%. We have not looked at the details, such as the location of the faults within the file, which is demands a more in depth investigation, or if the files are all "owned" by the same designer or not.

Our result was not what we expected, since we have had the assumption that e.g. more than 4% would be language pitfalls. This is why we felt reporting these findings would aid others in understanding more about the nature of faults. Our most important findings can be summarized as follows:

- Most of the faults were faults of omission or "missing path", meaning, not until execution on higher integration and system levels the lack of code was noted and the failure visible. A designer could clearly not find this, and the cause is most likely insufficient specifications available on lower level or lack of knowledge of the context of the code. A related fault class is spurious faults, where too much code is written, which might be overlapping or creating the problem.
- More than 34% of the corrections involve more than one file and the maximum of involved files for a correction is 25 files – the conclusion is that these faults are not possible to find on component level, and even 100% code-coverage on component level would not reveal the failure.
- Faults are much more complex than simple mistakes; often complicated logic confuses the developer. The semantics of faults are complex.
- Resource faults are not as complicated as e.g. static code faults when it comes to distribution of location.

5 Validation and Threats

This is a case study, which has low control over the environment, we have had some control over the measurements, but they were selected on the basis of possibility to gather (based on the labeling feature), which was random and outside our control. The replication is probably low of the experiment itself, but it should be possible to take any known failure/fault classes and do the same classification with a different outcome, depending on the software, process, organization and situation.

The main argument favoring our study is that, all failures and thus faults are from a live real system, and it is representative of the faults found and fixed, even if it is a not so huge sample. One possible validity threat is the selection of data is created based on the labeling function. The obvious way to perform a study on distribution of failures into fault classes would be to look at all faults, by comparing the difference between code versions. This is not a viable approach, since in systems, fault corrections, changes by adding new and modified code are mixed. Secondly, we have only focused on one small part of the system, since the main purpose was to identify the faults that we could make a copy of and re-inject the faults to use in controlled experiments. We investigated that for every reported failure, finding the actual fault, which represents the correction in the code, is still not completely accurate, since there is not a one to one relationship and this creates a problem in how to classify a fault. This is why we have shown two values, one based on the failure data reported, and one based on the adjusted software fault correction possible to re-inject in the code. The selection was made out of convenience, to find and create a sample to reason about. The failures corrections (labels) selection came from a wide community, of 65 designers from many countries and collected over a long time period and over many versions and changes in the system (3 years). We draw the conclusion that this fact lessens the internal threat, since the bias of interpretation has less impact, but cannot be disregarded.

Conclusion validity relates to subject selection, data collection, measurement reliability, and the validity of the statistical tests. These issues have been addressed in the design of the experiment, and we believe there is a disturbance in the data collection and measure reliability. We have used a nominal scale to classify our Trouble Reports. The classification is rather simple, which could be indicating problems with the internal validity. Classification into another system will yield a different result. Since our distribution is done with little insight of the software and no insight of history, process and organization and by and external party (thesis worker) the bias is minimized. There is no researcher bias put into the investigation, since it was done by a third party, and no guidance was given to what set of faults should be investigated, how the distribution and classification should be used or chosen. Thus, the researcher, who is familiar with some aspects of history, organization, process and software, which could indicate a bias, and a threat to the validity, have made the conclusions and verified the result.

Discussing the generalization of the result, we must look at external threats. The faults selected are on one particular product, but the nature of the product is such that it could probably be representative for lots of industrial software. We cannot conclude that that the result is generic since the distribution is dependent on organization, process, quality awareness, and a lot of other factors. However, earlier studies [12]

with similar results supports that for these types of systems the results could be generic and not an isolated result. We do think this is one example of a typical industry software fault distribution. Using our conclusions for any system might be pre-mature, but we suggest that this information can serve as an indication for complex middleware systems, operating systems, and similar large complex systems. All data is still available to pursue further studies.

6 Future Work

For natural reasons, a lot of the designer faults are found by designers themselves and are not registered. Depending on industry, designers are usually responsible for a large amount of the lower-level testing (unit, component, and lower level integration, and even some functional testing). Therefore, data on failures originates from independent test at different integration points, (levels), and from customers reports. We intend to investigate what faults and failures the designers themselves find.

Since we have noticed fault classifications often become failure or cause classifications, we aim to do a further study and make a fault classification (bug taxonomy) that are more useful for fault injection purposes. We believe we have already hinted on a structure (defining our classes and sub-classes), but feel we need to investigate and possibly expand these further. We also need to provide more clear rules for how to classify faults, so they cannot be classified in different classes depending on how the fault is interpreted. The large number of existing fault classifications will also be juxtaposed in such a new classification.

We would like to investigate automatic ways to classify legacy software, but that is not our primary concern, and we invite other researchers into a discussion about the feasibility of automatic classifications, as we have seen a published example of [13].

Our primary concern is to prepare code in many different versions, with real faults injected. These fault-injected code samples are intended to be used for evaluating test techniques in a controlled manner. We aim to inject faults into a system to minimize bias for one test technique over another. This require us to have a clear classification (with relation to the test technique), and there must be a variety of faults that behave (propagate) and is viewed differently, if we are to determine where a test technique would be more efficient. Our interest to find a set of faults that through execution behaved in different ways, and cause different failures in the software. The faults we have found, isolated, classified, and understood are to be re-injected in the code for better comprehension of how each one of them behaves when propagating to failures, as opposed to debugging. Of course, the variety of faults is more interesting, and we are to explore mutation techniques as a complement, and study how they behave and how their fault semantics could look like.

7 Discussions and Conclusions

We now return to the questions posed in the introduction and present our findings based on the collected data.

1. *What are real and important faults in software that have propagated to failures, and are fixed?*

The answer is clearly that faults of omission (38,3% of software faults), together with spurious faults (8,3 % of all software faults), shows that faults related to unclear specifications dominates among the real software faults found in the considered sub-system in a large complex middleware system. The main conclusion and contribution is the fact that the individual designer at component test level do not find these type of faults, and that they must be found at later stages in testing. It underlines how difficult it is for a designer to understand, define, specify and implement code in an environment of complexity, since not enough knowledge of context available. This is supported by looking at distribution of the faults over files, where 36% of these two classes of faults spans over more than one file.

2. *What is the distribution of failures and faults into classes?*

Our observations were as follows:

- 53% of all failures decided to be corrected are not relating to the software and are not possible to re-inject into the software
- Faults of omission (lack of code) is the dominating class among the software faults (38.3%)
- Computational/Logical (13.7%) and Data faults (12.7%) followed by Static Code faults (10.3%) are the next largest groups.
- Language pitfalls are only contributing with 4% of the software fault distribution

3. *Is there any other specific pattern of the faults and failures that would guide us into understanding the software process better?*

- We have found that as much as 19% of all faults are duplicates that still remain to be corrected (even after management has taken out duplicates). This shows that software failures are often expressed differently and not identified as duplicates until code is corrected.
- Software configuration management and build related faults together are contributing with as much as 15% of the failures.
- Conventional faults in software (computational/logical and functional faults, language pitfalls and static code faults) are only 39.6% of the software faults in this complex middleware system.

We think that these findings suggest where effort and cost should be placed. We know for a fact that this software had a targeted component test improvement (unit level) during the year 2005, which greatly improved the quality. We know that from the period 2003-2004 most of the trouble reports came from outside this software organization (applications, customers), but after the improvement most of the trouble reports originated from within the organization. We also know that the testing of these products have improved between 2003 and 2006, with only one test level in 2003, and now with more than 4 test levels. We can also see this reflecting in the number of duplications going down. We have noticed that viewing Trouble Reports (failures) over time, provides us information that many of the changes induced by the trouble reports that we considered were both re-designs for change and expansions purposes

and for fixing bad design, and these problems were more common earlier in the development than in the later stages of development. We suggest that changing software configuration and build system will have a great impact on the code (since our study indicated as much as 15% of faults are due to SCM). We think there is much information to be utilized from this study. We do believe it is humanly impossible to understand the entire system, and even if knowing context and teaching about it, this will only remedy a part of the problem, and that unit and component testing alone have no chance of finding a majority of these faults, which is of course already shown and evident. This study strengthens that evidence. Even if we strongly believe component testing is essential for complex systems that need to be robust, we must do testing on many other levels in the system to understand where the important faults hide. Testing is a support to the designer.

We were surprised that the fault categories that are the target of many static analysis tools had so low frequency, which leads us to suggesting a more cost–efficient way might be to work on specifications, understanding the software context, and test-set up. We believe that since the system is build in a "fail-safe" way with the aim to minimize impact of anything going wrong (by duplication of hardware, protocol resending, restarting etc) the impact of simple singular faults are often hidden or dormant. We believe that the improvement on component test are not reported in the same way in the trouble report system, since designers correct their own mistakes when they encounter them, rather than report them, which puts a hidden figure on these types of faults. Most of the component test faults (that is found by e.g. static analysis tools and component test) are not as visible in this study, but still do exist.

We have concluded why there is such a lack of information on code faults – and how good it is to do this analysis and really understand the information, since it gives Ericsson guidance on where efforts of improvements should be targeted. We understand the difficulty to gather this information if traceability of the code is not directly possible from the reported failure. The strength in our study is that it is unbiased to the code, and based solely on applying clear classification rules. We have of course encountered problems with classifications, how distinct the classes are, and how classifications should be applied. We intend to explore this in detail in a future work.

Acknowledgments. We would like to thank Ericsson for their support. We would also like to thank our sponsor The Knowledge Foundation, which via their SAVE-IT program made this research possible.

References

1. Eldh, S., Hansson, H., Punnekkat, S., Pettersson, A., Sundmark, D.: A Framework for Comparing Efficiency, Effectiveness and Applicability of Software Testing Techniques. In: Proc. TAIC, IEEE, New York (2006)
2. Juristo, N., Moreno, A.M., Vegas, S.: Reviewing 25 Years of Testing Technique Experiments. Journal of Empirical Softw. Eng., vol. 9(1-2), pp. 7–44. Springer, Heidelberg (2004)

3. Hyunsook, D., Elbaum, S., Rothermel, G.: Infrastructure support for controlled experimentation with software testing and regression testing techniques. In: Proc. Int. Symp. On Empirical Software Engineering, ISESE '04, pp. 60–70. ACM, New York (2004)
4. Apiwattanapong, T., Santelices, R., Chittimalli, P.V., Orso, A., Harrold, M.J.: Tata: MaTRIX: Maintenance-Oriented Test Requirements Identifier and Examiner. In: Proc. From TAIC, IEEE, New York (2006)
5. Basili, V.R., Selby, R.W.: Comparing the Effectiveness of Software Testing Strategies original 1985, revised dec. 87. In: Boehm, B., Rombach, H.D., Zelkowitz, M.V. (eds.) Foundations of Empirical Software Engineering, The Legacy of Victor R. Basili, Springer, Heidelberg (2005)
6. Myers, G.J.: A controlled experiment in program testing and code walkthroughs inspections, Comm. ACM, pp. 760–768 (September 1978)
7. Hetzel, W.C.: An experimental analysis of program verification methods, PhD dissertation, Univ. North Carolina, Chapel Hill (1976)
8. Chillarege, R., Inderpal, S., Bhandari, J.K., Chaar, M.J., Halliday, Moebus, D.S, Ray, B.K, Wong, M.-Y.: Orthogonal defect classification – a concept for in-process measurements. IEEE Trans. on Soft. Eng 18(11), 943–956 (1992)
9. Thane, H., Wall, A.: Testing Reusable Software Components in Safety-Critical Real-Time Systems, vol. 1(1-2) Artech House Publishers (2002)
10. Avižienis, A., Laprie, J.: Dependable computing: From concepts to design diversity. In: Proceedings of the IEEE, vol. 74, pp. 629–638 (May 1986)
11. Basili, V.R., Perricone, B.T.: Software errors and complexity: An empirical investigation. Communications of the ACM 27(1), 42–52 (1984)
12. Beizer, B.: Software Testing and Quality Assurance. Van Nostrand Reinhold electrical/computer science and engineering series. Van Nostrand Reinhold, NY (1984)
13. DeMillo, R.A., Maihur, A.P.: A grammar based fault classification scheme and its application to the classification of the errors of TEX. Technical Report SERC-TR-165-P, Purdue University, West Lafayette, IN 47907 (1995)
14. Endres, A.: An analysis of errors and their causes in system programs. Technical report, IBM Laboratory, Boebligen, Germany (1975)
15. Johnson, C., et al.: Guide to IEEE standard for classification for software anomalies. In: Technical report, IEEE Computer Society, Washington (1995)
16. Goodenough, J.B., Gerhart, S.L.: Toward a theory of test data selection. In: Proceedings of the international conference on Reliable software, pp. 493–510. ACM Press, New York, USA (1975)
17. Gray, J.: Why do computers stop and what can be done about it? Technical Report, vol. 85(7) Tandem Computers (1985)
18. Harrold, M.J., Offutt, A.J., Tewary, K.: An approach to fault modeling and fault seeding using the program dependence graph. Journal of Systems and Software 36(3), 273–296 (1997)
19. Howden, W.E.: Reliability of the path analysis testing strategy. IEEE Trans. on Software Engineering 2(3), 208–215 (1976)
20. Knuth, D.E.: The errors of TEX. Software Practice and Experience 7, 607–685 (1989)
21. Ishoda, S.: A criticism on the capture-and-recapture method for software reliability assurance. In: Proc. Soft. Eng, IEEE, New York (1995)
22. Ostrand, T.J., Weyuker, E.J, Bell, R.M.: Predicting the Location and Number of Faults in Large Software Systems. IEEE Trans. of Soft. Eng., vol. 31(4) (April 2005)

23. Perry, D.E., Steig, C.S.: Software faults in evolving a large, real-time system: a case study. In: Sommerville, I., Paul, M. (eds.) ESEC 1993. LNCS, vol. 717, pp. 48–67. Springer, Heidelberg (1993)
24. Kaner, C. Falk, J., Nguyen, H.Q: Testing Computer Software. 2nd edn. International Thomson Computer Press (1993)
25. Vaidyanathan, K., Kishor, S., Trivedi, A.: A comprehensive model for software rejuvenation. IEEE Trans. on Dependable and Secure Computing 2(2), 124–137 (2005)
26. Zeil, S.J.: Perturbation techniques for detecting domain errors. IEEE Transactions on Software Engineering 15(6), 737–746 (1989)
27. Mohaghegi, P., Conradi, R., Borretzen, J.A.: Revisiting the problem of Using Problem Reports for Quality Assessments, WQSA, ICSE (2006)
28. Henningsson, K., Wohlin, C.: Assuring fault classification agreement – An Empirical Evaluation. In: Proc. of ISESE'04, IEEE, New York (2004)
29. Offut, A.J., Huffman- Hayes, J.: A Semantic Model of Program Faults. In: Proceedings of ISSTA, pp. 195–200 (1996)
30. Damm, L.O, Lundberg, L., Wohlin, C.: Fault-Slip Through - a concept for measuring the efficiency of the test process. Journal of Software Process: Improvements and Practice, vol. 11(1), pp. 47–59. John Wiley and Sons, New York (2006)
31. Murphy, B., Garzia, M., Suri, N.: Closing the Gap in Failure Analysis. Workshop on Applied SW Reliability-DSN (2006)
32. Andrews, J.H., Briand, L.C., Labiche, Y.: Is Mutation an Appropriate Tool for Testing Experiments? In: ICSE 2005, ACM, New York (2005)
33. Frankl, P.G., Iakounenko, O.: Further Empirical Studies of test Effectiveness. In: Proc. 6th ACM SIGSOFT International Symposium on Foundations of Software Engineering, Orlando, FL, USA, pp. 153–162 (1998)
34. Vokolos, F.I., Frankl, P.G.: Empirical evaluation of the textual differencing regression testing technique. In: Proc. IEEE Int. Conference on Soft. Maint. USA, pp. 44–53 (1998)
35. IEEE Std. 1044 -1993, Standard for classification for software anomalies. IEEE (1993)