Software Faults in Evolving a Large, Real-Time System: a Case Study

Dewayne E. Perry and Carol S. Stieg

[1] AT&T Bell Laboratories, 600 Mountain Avenue, Murray Hill, NJ 07974 USA
[2] AT&T Bell Laboratories, 263 Shuman Blvd, Naperville, NJ 60566 USA

Abstract. We report the results of a survey about the software faults encountered during the testing phases in evolving a large real-time system. The survey was done in two parts: the first part surveyed all the faults that were reported and characterized them in terms of general categories; the second part resurveyed in depth the faults found in the design and coding phases. For the first part, we describe describe the questionaire, report the general faults found, and characterize the requirements, design and coding faults by the testing phases in which they were found and by the time they were found during the testing interval. For the second part, we describe the questionaire used to survey the design and coding faults, report the faults that occurred, how difficult they were to find and fix, what their underlying causes were (that is, what their corresponding errors were), and what means might have prevented them from occurring. We then characterize the results in terms of interface and implementation faults.

1 Introduction

It is surprising that so few software fault studies have appeared in the software engineering literature, especially since monitoring our mistakes is one of the fundamental means by which we improve our process and product. This is particularly true for the development of large systems. In preceding work [13, 14], Perry and Evangelist reported the prevalence of interface faults as a major factor in the development and evolution of a large real-time system (68% of the faults). One of the main purposes of that software fault study was to indicate the importance of tools (such as the Inscape Environment [16]) that manage interfaces and the dependencies on those interfaces.

Prior to this work, Endres [7], Schneidewind and Hoffman [19], and Glass [8] reported on various fault analyses of software development, but did not delineate interface faults as a specific category. Thayer, Lipow and Nelson [21] and Bowen [5] provide extensive categorization of faults, but with a relatively narrow view of interface faults. Basili and Perricone [2] offer the most comprehensive study of problems encountered in the development phase of a medium-scale system, reporting data on the fault, the number of components affected, the type of the fault, and the effort required to correct the fault. Interface faults were the largest class of faults (39% of the faults).

We make two important contributions in this case study. First, we present software fault data on evolutionary development [16], not on initial development as previous studies have done. Second, we use a novel approach in which we emphasize the cost of the finding (i.e., reproducing) and fixing faults and the means of preventing them.

In section 2, we provide the background for the study, describing the system in general terms, and the methodology employed in evolving the system. In section 3, we describe our experimental strategy and the approach we used in conducting the survey. In section 4, we report the overall Modification Request (MR) survey, providing first a summary of the questionnaire, then a summary of the results, and finally some conclusions. In section 5, we present the design and coding fault survey, providing first a summary of the questionnaire, then a discussion of the analysis, and finally a summary relating the results to interface faults. In section 6, we present conclusions and recommendations.

2 Background

The system discussed in this paper is a very large [3] scale, distributed, real-time system written in the C programming language in a Unix-based, multiple machine, multiple location environment.

The organizational structure is typical with respect to AT&T projects for systems of this size and for the number of people in each organization. Not surprisingly, different organizations are responsible for various parts of the system development: requirements specification; architecture, design, coding and capability testing; system and system stability testing; and alpha testing.

The process of development is also typical with respect to AT&T projects of this size. Systems Engineers prepare informal and structured documents defining the requirements for the changes to be made to the system. Designers prepare informal design documents that are subjected to formal reviews by three to fifteen peers depending on the size of the unit under consideration. The design is then broken into design units for low level design and coding. The products of this last phase are subjected both to formal code reviews by three to five reviewers and to low level unit testing. As components become available, integration and system testing is performed until the system is completely integrated.

The release considered here is a "non-initial" release —one that can be viewed as an arbitrary point in the evolution of this class of systems. Because of the size of the system, the system evolution process consists of multiple, concurrent releases —that is, while the release dates are sequential, a number of releases proceed concurrently in differing phases. This concurrency accentuates the inter-release dependencies and their associated problems. The magnitude of the changes (approximately 15-20% new code for each release) and the general make-up of the changes (bug-fixes, improvements, and new functionality, etc.) are generally uniform across releases. It is because of these two facts that we

[3] By "very large", we mean a system of 1,000,000 NCSL or more [4]. AT&T has a wide variety of such very large systems.

consider this study to provide a representative sample in the life of the project. This relative uniformity of releases contrasts with Lehman and Belady [11] where releases alternated between adding new functionality and fixing existing problems.

Faults discovered during testing phases are reported and monitored by a modification request (MR) tracking system (such as for example, CMS [18]). Access to source files for modification is possible only through the tracking system. Thus all source change activity is automatically tracked by the system. This activity includes not only repairs but enhancements and new functionality as well. It should be kept in mind, however, that this fault tracking activity occurs only during the testing and released phases of the project, not during the architecture, design and coding phases. Problems encountered during these earlier phases are resolved informally without being tracked by the MR system.

3 Survey Strategy

The goal of this study was to gain insight into the current process of system evolution by concentrating on a representative release of a particular system. The approach we used is that of surveying, by means of prepared questionnaires, those who "owned" the MR at the time it was closed. We conducted two studies: first, we surveyed the complete set of faults; second, we resurveyed the largest set of faults (i.e., the design and coding faults) in more depth.

It was mandated by management that the survey be non-intrusive, anonymous and strictly voluntary. The questionaire was created by the authors working with a group of developers involved in the study. It was reviewed by an independent group of developers who were also part of the subject development. What we were not able to do was to validate any of the results [3] and hence cannot assess the accuracy of the resulting data.

68% of the questionaires were returned in both parts of the study. In each case, the sample size was very large —sufficiently large to justify the precision that we use in the remainder of the paper. While there might be some questions about the representativeness of the responses, we know of no factors that would skew the results significantly [1].

4 Overall Survey

There were three specific purposes in the initial, overall survey:

- to determine what kinds of general problems (which we report here) and what kinds of specific application problems (which we do not report because of their lack of generality) were found during the preparation of this release;
- to determine how the problem was found (that is, in which testing phase);
- to determine when the problem was found.

In the discussion that follows, we first present a summary of the questionnaire, summarize our results, and draw some conclusions.

4.1 Questionnaire

The first survey questionnaire has two main components: the determination of the fault reported in the MR and the testing phase in which the fault was found. In determining the fault, two aspects were of importance: first, the development phase in which the fault was introduced, and second, the particular type of the fault. Since the particular type of fault reported at this stage of the survey tended to be application or methodology specific, we have emphasized the phase-origin nature of the fault categorization. The general fault categories are as follows:

- *Previous* —residual problems left over from previous releases;
- *Requirements* —problems originating during the requirements specification phase of development;
- *Design* —problems originating during the architectural and design phases of development;
- *Coding* —problems originating during the coding phases of development;
- *Testing Environment* —problems originating in the construction or provision of the testing environment (for example, faults in the system configuration, static data, etc);
- *Testing* —problems in testing (for example, pilot faults, etc);
- *Duplicates* —problems that have already been reported;
- *No problems* —problems due to misunderstandings about interfaces, functionality, etc., on the part of the user;
- *Other* —various problems that do not fit neatly in the preceding categories such as hardware problems, etc.

The other main component of the survey concerned the phase of testing that uncovered the fault. The following are the different testing phases.

- *Capability Test* (CT) —testing isolated portions of the system to ensure proper capabilities of that portion.
- *System Test* (ST) —testing the entire system to ensure proper execution of the system as a whole in the laboratory environment.
- *System Stability Test* (SS) —testing with simulated load conditions in the laboratory environment for extended periods of time.
- *Alpha Test* (AT) —live use of the release in a friendly user environment.
- *Released* (RE) —live use. However, in this study, this data refers not to this release, but the previous release. Our expectation is that this provides a projection of the fault results for this release.

The time interval during which the faults were found (that is, when the MRs were initiated) was retrieved from database of the MR tracking system.

Ideally, the testing phases occur sequentially. In practice, however, due to the size and complexity of the system, various phases overlap. The overlap is due to several specific factors. First, various parts of the system are modified in parallel. This means that the various parts of the system are in different states at any one time. Second, the iterative nature of evolution results in recycling back through previous phases for various parts of the system. Third, various testing phases are begun as early as possible, even though it is known that that component may be incomplete.

Looked at in one way, testing proceeds in a hierarchical manner: testing is begun with various pieces, then subsystems and finally integrating those larger parts into the complete system. It is a judgment call as to when different parts of the system move from one phase to the next determined primarily by the percentage of capabilities incorporated and the number of tests executed. Looked at in a slightly different way, testing proceeds by increasing the system's size and complexity, while at the same time increasing its load and stress.

4.2 Results

We present the summary of each fault category and discuss some of the main issues that stem from these results. Next we summarize the requirements, design and coding faults first as found by testing phase and then as found by time interval.

Table 1. Summary of Responses

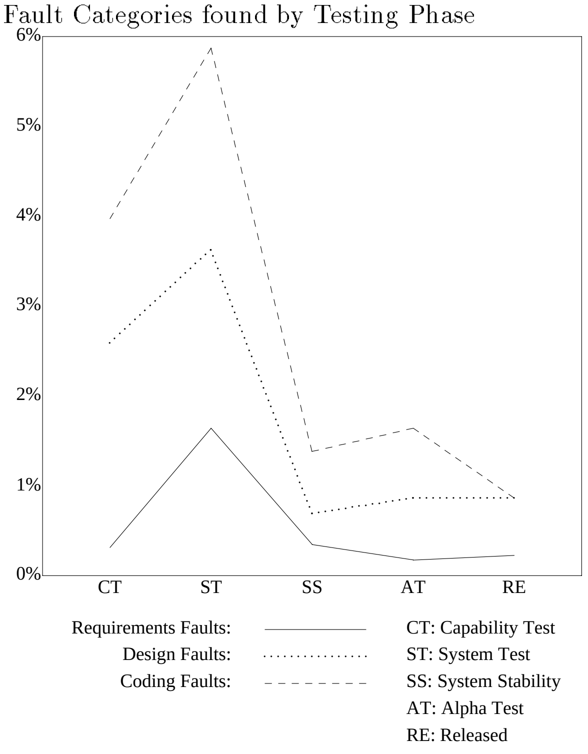
MR Categories	Proportion
Previous	4.0%
Requirements	4.9%
Design	10.6%
Coding	18.2%
Testing Environment	19.1%
Testing	5.7%
Duplicates	13.9%
No problems	15.9%
Other	7.8%

Responses. Table 1 summarizes the frequency of the MRs by category. "Previous" problems are those which existed in precious releases but only surfaced in the current release. They indicate the difficulty in finding some faults and the difficulty in achieving comprehensive test coverage. The MRs representing the earlier part of the development or evolution process (that is, those representing requirements, design and coding) are the most significant, accounting for approximately 33.7% of the MRs.

The next most significant subset of MRs were those that concern testing (the testing environment and testing categories) —24.8% of the MRs. It is not surprising that a significant number of problems are encountered in testing a large and complex real-time system where conditions have to be simulated to represent the "real-world" in a laboratory environment. First, the testing environment itself is a large and complex system that must be tested. Second, as the real-time system evolves, so must the laboratory test environment evolve.

"Duplicate" and "No Problem" MRs are another significant subset of the data —28.9%. Historically, they have been considered to be part of the overhead. The

Fig. 1. Fault Categories found by Testing Phase



"duplicate" MRs are in large part due to the inherent concurrency of activities in a large-scale project and, as such, are difficult to eliminate.

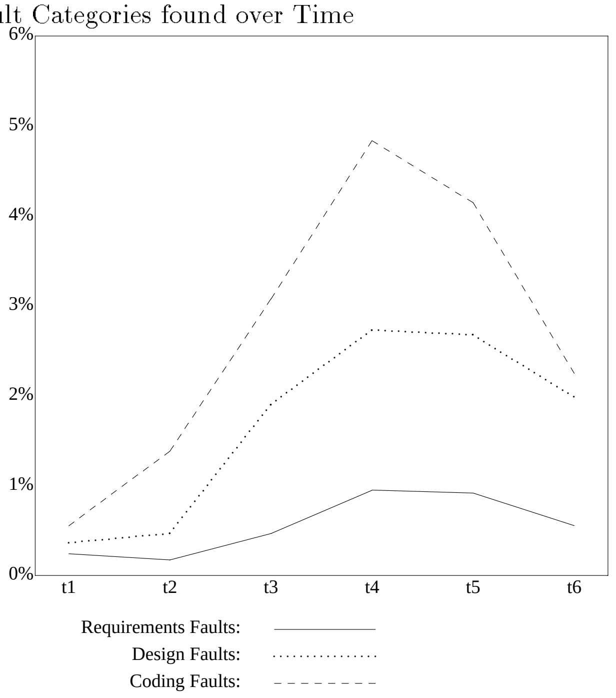
Results by Testing Phase and Time. We focus on the early part of the software process because that is where the most MRs occur and, accordingly, where close attention should yield the most results. For this reason, we present the requirements, design and coding faults distributed by testing phase.

For the requirements, design and coding fault categories, Figure 1 shows the percentage of MRs found during each testing phase. There are two important observations. First, system test (ST) was the source of most of the MRs in each category; capability testing (CT) was the next largest source. Second, all testing phases found MRs of each fault category.

We note that there are two reasons why design and requirements faults continue to be found throughout the entire testing process. First, requirements often change during the long development interval represented here. Second, informal requirement and design documents lack precision and completeness (a general problem in the current state-of-practice rather than a project-specific problem).

The data present in figure 2 represents the same MRs as in figure 1, but displayed according to when they were found during the testing interval. The

Fig. 2. Fault Categories found over Time



time values here are fixed (and relatively long) intervals. From the shape of the data, it is clear that System Testing overlaps interval t4.

For the requirements, design and coding fault categories over time, Figure 2 shows that all fault types peaked at time t4, and held through time t5, except for the coding faults which decreased. The two figures are different because there is non-trivial mapping between testing phase and calendar time.

4.3 Summary

The following general observations may be drawn from this general survey of the problems encountered in evolving a very large real-time system:

– all faults occurred throughout all the testing phases, and
– the majority of faults were found late in the testing interval.

These observations are limited by the fact that the tracking of MRs is primarily a testing activity. It would be extremely useful to observe the kinds and frequencies of faults that exists in the earlier phases of the project. Moreover, it would be beneficial to incorporate ways of detecting requirements and design faults into the existing development process.

5 Design/Code Fault Survey

As a result of the general survey, we decided to resurvey the design and coding MRs in depth. The following were the goals we wanted to achieve in this part of the study:

- determine the kinds of faults that occurred in design and coding;
- determine the difficulty both in finding or reproducing these faults and in fixing them;
- determine the underlying causes of the faults; and
- determine how the faults might have been prevented.

There were two reasons for choosing this part of the general set of MRs. First, it seemed to be exceedingly difficult to separate the two kinds of faults. Second, catching these kinds of faults earlier in the process would provide a significant reduction in overall fault cost —that is, the cost of finding faults before system integration is significantly less than finding them in the laboratory testing environment. Our internal cost data is consistent with Boehm's [4]. Thus, gaining insight into these problems will yield significant and cost beneficial results.

In the three subsections that follow, we summarize the survey questionnaire, present the results of our analysis, and summarize our findings with regard to interface and implementation faults.

5.1 Questionnaire

For every MR, we asked for the following information: the actual fault, the difficulty of finding and fixing the fault, the underlying cause, the best means of either preventing or avoiding the problem, and their level of confidence in their responses.

Fault Types. For this fault, consider the following 22 possible types and select the one that most closely applies to the cause of this MR.

1. *Language pitfalls* —e.g., the use of "=" instead of "= =".
2. *Protocol* —violated rules about interprocess communication.
3. *Low-level logic* —e.g., loop termination problems, pointer initialization, etc.
4. *CMS complexity* —e.g., due to change management system complexity.
5. *Internal functionality* —either inadequate functionality or changes and/or additions were needed to existing functionality within the module or subsystem.
6. *External functionality* —either inadequate functionality or changes and/or additions were needed to existing functionality outside the module or subsystem.
7. *Primitives misused* —the design or code depended on primitives which were not *used* correctly.

8. *Primitives unsupported* —the design or code depended on primitives that were not adequately developed (that is, the primitives did not work correctly).
9. *Change coordination* —either did not know about previous changes or depended on concurrent changes.
10. *Interface complexity* —interfaces were badly structured or incomprehensible.
11. *Design/Code complexity* —the implementation was badly structured or incomprehensible.
12. *Error handling* —incorrect handling of, or recovery from, exceptions.
13. *Race conditions* —incorrect coordination in the sharing of data.
14. *Performance* —e.g., real-time constraints, resource access, or response time constraints.
15. *Resource allocation* —incorrect resource allocation and deallocation.
16. *Dynamic data design* —incorrect design of dynamic data resources or structures.
17. *Dynamic data use* —incorrect *use* of dynamic data structures (for example, initialization, maintaining constraints, etc.).
18. *Static data design* —incorrect design of static data structures (for example, their location, partitioning, redundancy, etc.).
19. *Unknown Interactions* —unknowingly involved other functionality or parts of the system.
20. *Unexpected dependencies* —unexpected interactions or dependencies on other parts of the system.
21. *Concurrent work* —unexpected dependencies on concurrent work in other releases.
22. *Other* —describe the fault.

Ease of Finding or Reproducing the Fault. The MR in question is to be ranked according to how difficult it was to reproduce the failure and locate the fault.

1. *Easy* —could produce at will.
2. *Moderate* —happened some of the time (intermittent).
3. *Difficult* —needed theories to figure out how to reproduce the error.
4. *Very Difficult* —exceedingly hard to reproduce.

Ease of Fixing the Fault. For each MR, how much time was needed to design and code the fix, document and test it. [4]

1. *Easy* —less than one day
2. *Moderate* —1 to 5 days
3. *Difficult* —6 to 30 days
4. *Very difficult* —greater than 30 days

[4] Note that what would be an easy fix in a single programmer system takes considerably more time in a large, multi-person project with a complex laboratory test environment.

Underlying Causes. Because the fault may be only a symptom, provide what you regard to be the underlying root cause for each problem.

1. *None given* —no underlying causes given.
2. *Incomplete/omitted requirements* —the source of the fault stemmed from either incomplete or unstated requirements.
3. *Ambiguous requirements* —the requirements were (informally) stated, but they were open to more than one interpretation. The interpretation selected was evidently incorrect.
4. *Incomplete/omitted design* —the source of the fault stemmed from either incomplete or unstated design specifications.
5. *Ambiguous design* —the design was (informally) given, but was open to more than one interpretation. The interpretation selected was evidently incorrect.
6. *Earlier incorrect fix* —the fault was induced by an earlier incorrect fix (that is, the fault was not the result of new development).
7. *Lack of knowledge* —there was something that I needed to know, but did not know that I needed to know it.
8. *Incorrect modification* —I suspected that the solution was incorrect, but could not determine how to correctly solve the problem.
9. *Submitted under duress* —the solution was submitted under duress, knowing that it was incorrect (generally due to schedule pressure, etc).
10. *Other* —describe the underlying cause.

Means of Prevention. For this fault, consider possible ways to prevent or avoid it and select the most useful or appropriate choice for preventing or avoiding the fault.

1. *Formal requirements* —use precise, unambiguous requirements (or design) in a formal notation (which may be either graphical or textual).
2. *Requirements/Design templates* —provide more specific requirements (or design) document templates.
3. *Formal interface specifications* —use a formal notation for describing the module interfaces.
4. *Training* —provide discussions, training seminars, and formal courses.
5. *Application walk-throughs* —determine, informally, the interactions among the various application specific processes and data objects.
6. *Expert person/documentation* —provide an “expert” person or clear documentation when needed.
7. *Design/code currency* —keep design documents up to date with code changes.
8. *Guideline enforcement* —enforce code inspections guidelines and the use of static analysis tools such as lint.
9. *Better test planning* —provide better test planning and/or execution (for example, automatic regression testing).
10. *Others* —describe the means of prevention.

Confidence Levels. Confidence levels requested of the respondents were: *very high, high, moderate, low* and *very low* . We discarded the small number of responses (6%) that had a confidence level of either low or very low.

5.2 Analysis

Table 2. Chi-Square Analysis Summary

Variables	Degrees of Freedom	Total Chi-Square	p
Find, Fix	6	51.489	.0001
Fault, Find	63	174.269	.0001
Fault, Fix	63	204.252	.0001
Cause, Find	27	94.493	.0001
Cause, Fix	27	55.232	.0011
Fault, Cause	189	403.136	.0001
Prevention, Find	27	41.021	.041
Prevention, Fix	27	97.886	.0001
Fault, Prevention	189	492.826	.0001
Cause, Prevention	81	641.417	.0001

To understand the relationships between the faults, the effort to find and fix them, their underlying causes, and their means of prevention, the results of the survey were cross tabulated and then subjected to chi-square analysis. Table 2 provides a summary of the chi-square analysis.

To identify relationships, we tested against the hypothesis that each member of pairs were independent of each other. To establish whether any relationship existed we used the Chi-Square test [20]. Table 2 provides the results of that test. The numbers show that the relationships are statistically significant at the level indicated by the value of p. Note that that in all cases the relationships are statistically significant.

The relationship between the means of prevention and the ease of finding the fault coming closest to being independent (because $p = .041$; if p had been .05 or greater, we would say they were independent). The relationship between the means of prevention and the underlying causes is the most significantly interdependent (because the total chi-square is so large).

In earlier work, Perry and Evangelist [13,14] received comments about the cost of faults —specifically, were the interface faults the easy ones or the hards ones. It is with these questions in mind that we included the questions about the effort to find and fix the faults. We have devised simple weighting measures for finding and fixing faults that are meant to be indicative rather than definitive. These measures are intended to support a relative, not absolute, comparison of the faults with each other.

To estimate the effort to find a fault, we determine the weight by multiplying the proportion of observed values for each fault by 1, 2, 3 and 4 for each effort category, respectively, and sum the results. For example, if a fault was easy to find in 66% of the cases, moderate in 23%, difficult in 11%, and very difficult in

1%, the weight is 146 = (66 * 1) + (23 * 2) + (10 * 3) + (1 * 4). The higher the weight, the more effort required to find the fault.

To estimate the effort to fix a fault, we determine the weight by multiplying the proportion of observed values for each fault by 1, 3, 15, and 30. We chose these values because they are a reasonable approximation to the average length of time to fix each set of faults. Using the same proportions as above, the corresponding fix weight would be 315 = (66 * 1) + (23 * 3) + (10 * 15) + (1 * 30).

To compute the effort-adjusted frequencies of each fault, we multiply the number of occurrences of each fault by its weight and divide by the total weighted number of occurrences.

We first consider the difficulty of finding and fixing the faults. We then discuss the faults and the cost of finding and fixing them. Next we consider the underlying causes and the means of prevention, correlate the faults and their effort measures with underlying causes and means of prevention, and then correlate underlying causes and means of prevention. Finally, we divide the faults into interface and implementation categories and compare them in terms of difficulty to find and fix, their underlying causes and their means of prevention.

Finding and Fixing Faults. 91% of the faults were easy to moderate to find; 78% took less than 5 days to fix. In general, the easier to find faults were easier to fix; the more difficult to find faults were more difficult to fix as well. There were more faults that were easy to find and took less than one day to fix than were expected by the chi-square analysis. Interestingly, there were fewer than expected easy to find faults that took 6 to 30 days to fix.

Table 3. Summary of Find/Fix Effort

Find/Fix Effort	$\leq$ 5 Days	$\geq$ 6 Days
easy/moderate	72.5%	18.4%
difficult/very difficult	5.9%	3.2%

While the coordinates of the effort to find and fix the faults are not comparable, we note that the above relationship between them is suggestive. Moreover, it seems counter to the common wisdom that says "once you have found the problem, it is easy to fix it". There is a significant number of "easy/moderate to find" faults that require a relatively long time to fix.

Faults. Table 4 shows the fault types of the MRs as ordered by their frequency in the survey independent of any other factors. We also show the effort weights,the effort ranks, and the weight-adjusted frequencies for finding and fixing these faults.

The first 5 fault categories account for 60% of the observed faults. That "internal functionality" is the leading fault by such a large margin is somewhat surprising; that "interface complexity" is such a significant problem is not surprising at all. However, that the first five faults are leading faults is consistent with the nature of the evolution of the system. Adding significant amounts of

Table 4. Faults Ordered by Frequency

Fault	Description	Observed	Find Weight	Find Rank	Find Adj'd	Fix Weight	Fix Rank	Fix Adj'd
5	internal functionality	25.0%	134	14	23.4%	414	13	18.7%
10	interface complexity	11.4%	145	11	11.6%	607	10	12.6%
20	unexpected dependencies	8.0%	124	19	7.2%	786	4	11.3%
3	low-level logic	7.9%	132	15	7.3%	245	17	3.5%
11	design/code complexity	7.7%	164	3	8.8%	904	3	12.6%
22	other	5.8%	131	16	5.5%	499	12	5.4%
9	change coordination	4.9%	150	10	5.1%	394	14	3.5%
21	concurrent work	4.4%	131	16	4.0%	661	9	5.2%
13	race conditions	4.3%	209	1	6.3%	709	7	5.5%
6	external functionality	3.6%	139	13	3.5%	682	8	4.4%
1	language pitfalls	3.5%	141	12	3.3%	244	18	1.5%
12	error handling	3.3%	163	4	3.7%	717	6	4.3%
7	primitive's misuse	2.4%	120	20	2.0%	520	11	2.2%
17	dynamic data use	2.1%	158	6	2.3%	392	15	1.5%
15	resource allocation	1.5%	161	5	1.7%	1326	2	3.6%
18	static data design	1.0%	100	21	.7%	200	19	.4%
14	performance	.9%	199	2	1.3%	1402	1	2.3%
19	unknown interactions	.7%	157	7	.8%	785	5	1.0%
8	primitives unsupported	.6%	151	9	.6%	200	19	.2%
2	protocol	.4%	125	18	.2%	250	16	.2%
4	CMS complexity	.3%	100	21	.2%	166	21	.1%
16	dynamic data design	.3%	157	7	.3%	166	21	.1%

new functionality to a system easily accounts for problems with "internal functionality", "low-level logic" and "external functionality".

The fact that the system is a very large, complicated real-time system easily accounts for the fact that there are problems with "interface complexity", "unexpected dependencies" and design/code complexity", "change coordination" and "concurrent work".

C has well-known "language pitfalls" that account for the rank of that fault in the middle of the set. Similarly, "race conditions" are a reasonably significant problem because of the lack of suitable language facilities in C.

That "performance" faults are a relatively insignificant is probably due to the fact that this is not an early release of the system. Their first ranking fix weight is consistent with our intuition that they are extremely difficult to fix.

Finding and Fixing Faults. We find the weighted ordering here affirms our intuition of how difficult these faults might be to find. Typically, performance faults and race conditions are very difficult to isolate and reproduce. We would expect that "code complexity" and "error handling" faults would also be difficult to find and reproduce.

When we inspect the chi-square test for faults and find effort, we notice that "internal functionality", "unexpected dependencies" and "other" tended to be easier to find than expected. "Code complexity" and "performance" tended to be harder to find than expected. There tended to be more significant deviations where the sample population was larger.

Adjusting the frequency by the effort to find the faults results in only a slight shift in the ordering of the faults. "Internal functionality", "code complexity", and "race conditions" change slightly more than the rest of the faults.

Adjusting the frequency by the effort to fix the fault causes some interesting shifts in the ordering of the faults. "Language pitfalls", "low-level logic", and "internal functionality" drop significantly in their relative importance. This coincides with one's intuition about these kinds of faults —i.e., they are easy to fix. "Design/code complexity", "resource allocation", and "unexpected dependencies" rise significantly in their relative importance; "interface complexity", "race conditions", and "performance" rise but not significantly so.

The top four faults account for 55% of the effort expended to fix all the faults and 51% of the effort to find them, but represent 52% of the faults by observed frequency. Collectively, they are somewhat harder to fix than rest of the faults and slightly easier to find. We again note that while the two scales are not strictly comparable, the comparison is an interesting one none-the-less.

When we inspect the chi-square test for faults and find effort, we notice that "language pitfalls", and "low-level logic" took fewer days to fix than expected. "Interface complexity" and "internal functionality" took 1 to 30 days more often than expected, while "design/code complexity" and "unexpected dependencies" took longer to fix (that is, 6 to over 30 days) than expected. These deviations reenforce our weighted assessment of the effort to fix the faults.

Underlying Causes. Table 5 shows the underlying causes of the MRs as ordered by their frequency in the survey independent of any other factors, the effort weights, effort ranks, and effort-adjusted frequencies for finding and fixing the faults with these underlying causes.

Table 5. Underlying causes of Faults

Cause	Description	Observed	Find Weight	Find Rank	Find Adj'd	Fix Weight	Fix Rank	Fix Adj'd
4	incomplete/omitted design	25.2%	139	8	24.6%	653	3	29.7%
1	none given	20.5%	150	2	21.5%	412	10	15.2%
7	lack of knowledge	17.8%	135	9	16.8%	525	8	16.8%
5	ambiguous design	9.8%	141	6	9.6%	464	9	8.1%
6	earlier incorrect fix	7.3%	147	4	7.5%	544	7	7.1%
9	submitted under duress	6.8%	158	1	7.5%	564	6	6.9%
2	incomplete/omitted req's	5.4%	143	5	5.4%	698	2	6.8%
10	other	4.1%	148	3	4.2%	640	4	4.7%
3	ambiguous requirements	2.0%	140	7	2.9%	940	1	3.4%
8	incorrect modification	1.1%	109	10	.8%	588	5	1.3%

Weighting the underlying causes by the effort to find or reproduce the faults for which these are the underlying causes produces almost no change in either the ordering or in the relative proportion of the underlying causes.

With respect to the relative difficulty in finding the faults associated with the underlying causes, the resulting ordering is particularly non-intuitive: the MRs with no underlying cause are the second most difficult to find; those submitted under duress are the most difficult to find.

Weighting the underlying causes by the effort to fix the faults represented by the underlying causes yields a few shifts in the proportion of effort: “incomplete/omitted design” increased significantly, “unclear requirements” and “incomplete/omitted requirements” increased less significantly; “none” decreased significantly, “unclear design” and “other” decreased less significantly. However, the relative ordering of the various underlying causes is approximately the same.

The relative weighting of the effort to fix these kinds of underlying causes seems to coincide with one’s intuition very nicely.

When we inspect the chi-square test for fix effort and underlying causes, we notice faults caused by “none given” tended to take less time to fix than expected, while faults caused by “incomplete/omitted design” and “submitted under duress” tended to take more time to fix than expected.

The high proportion of “none given” as an underlying cause requires some explanation. One of the reasons for this is that faults such as “language pitfalls”, “low-level logic”, “race conditions” and “change coordination” tend to be both the fault and the underlying cause (7.8% —or 33% of the in the “none given” underlying cause category). In addition, one could easily imagine that some of the faults such as “interface complexity” and “design/code complexity” could also be considered both the fault and the underlying cause (3.3% —or 16% of the faults in the “none given” underlying cause category). On the other hand, we were surprised that no cause was given for a substantial part of the “internal functionality” faults (3.7% —or 18% of the faults in the “none given” category). One would expect there to be some underlying cause for that particular fault.

Means of Prevention. Table 6 shows the suggested means of prevention of the faults as ordered by their occurrence independent of any other factors, the effort weights, effort ranks, and effort-adjusted frequencies for finding and fixing the faults to which these means of prevention are applicable. We note that the various means of prevention are by no means independent or non-overlapping. Moreover, the means selected may well reflect a particular approach of the responder in selecting one means over another (for example, see the discussion below about formal versus informal means of prevention).

It is interesting to note that the application-specific means of prevention (“application walk-throughs”) is considered the most effective means of prevention. This selection of application walk-throughs as the most useful means of error prevention appears to confirm the observation of Curtis, Krasner and Iscoe [6] that a thin spread of application knowledge is one of the most significant problem in building large systems.

Further it is worth noting that informal means of prevention rank higher than formal ones. On the one hand, this may reflect the general bias in the United States against formal methods. On the other hand, the informal means are a non-technical solution to providing the information that may be supplied by the formal representations (and which provide a more technical solution with higher

Table 6. Means of Error Prevention

Means	Description	Observed	Find Weight	Find Rank	Find Adj'd	Fix Weight	Fix Rank	Fix Adj'd
5	appl'n walk-throughs	24.5%	140	8	24.0%	438	8	18.9%
6	expert person/doc'n	15.7%	145	2	15.9%	706	3	19.6%
8	guideline enforcement	13.3%	154	1	14.3%	389	10	9.1%
2	req's/design templates	10.0%	129	10	9.1%	654	5	11.6%
9	better test planning	9.9%	145	2	10.1%	401	9	7.0%
1	formal requirements	8.8%	144	6	8.8%	740	2	11.4%
3	formal interface spec's	7.2%	142	7	7.1%	680	4	8.6%
10	other	6.9%	145	2	7.0%	517	6	8.7%
4	training	2.2%	145	2	2.2%	1016	1	3.9%
7	design/code currency	1.5%	140	8	1.5%	460	7	1.2%

adoption costs).

The level of effort to find the faults for which these are the means of prevention does not change the order found in the table above, with the exception of "requirements/design templates" which seems to apply to the easier to find faults and "better test planning" which seems to apply more to somewhat harder to find faults.

When we inspect the chi-square test for find effort and means of prevention, we notice that the relationship between finding faults and preventing them is the most independent of the relationships discussed here. "Application walk-throughs" applied to faults that were marginally easier to find than expected, while "guideline enforcement" applied to faults that were less easy to find than expected.

When we inspect the chi-square test for fix effort and means of prevention, it is interesting to note that the faults considered to be prevented by training are the hardest to fix. Formal methods also apply to classes of faults that take a long time to fix.

Effort-adjusting the frequency by fix effort yields a few shifts in proportion: "application walk-throughs", "guideline enforcement" and "better test planning" decreased in proportion; "expert person/documentation" and "formal requirements" increased in proportion, "formal interface specifications" and "other" less so. As a result, the ordering changes slightly to faults 6, 5, 2, 1, 8, 10, 3, 9, 4, 7: "expert person/documentation" and " formal requirements" are weighted significantly higher; "requirements/design templates", "formal interface specifications", "training", and "other" are less significantly higher.

When we inspect the chi-square test for faults and means of prevention, we notice that faults prevented by "application walk-throughs", "guideline enforcement", and "other" tended to take fewer days to fix than expected, while faults prevented by "formal requirements", requirements/design templates" and "expert person/documentation" took longer to fix than expected.

Underlying Causes and Means of Prevention. It is interesting to note that in the chi-square test for underlying causes and means of prevention there are a significant number of deviations (that is, there is a wider variance between the actual values and the expected values in correlating underlying causes and means of prevention) and that there does not appear to be much statistical structure. This indicates that there are strong dependencies between the underlying causes and their means of prevention. Intuitively, this type of relationship is just what we would expect.

5.3 Interface Faults versus Implementation Faults

The definition of an interface fault that we use here is that of Basili and Perricone [2] and Perry and Evangelist [13, 14]: interface faults are "those that are associated with structures existing outside the module's local environment but which the module used". Using this definition, we roughly characterize "language pitfalls" (1), "low-level logic" (3), "internal functionality" (5), "design/code complexity" (11), "performance" (14), and "other" (22) as implementation faults. The remainder are considered interface faults. We say "roughly" because there are some cases where the implementation categories may contain some interface problems —e.g., some of the "design/code complexity" faults were considered preventable by formal interface specifications.

Table 7. Interface/Implementation Fault Comparison

	Interface	Implementation
frequency	49%	51%
find weighted	50%	50%
fix weighted	56%	44%

Interface faults occur with slightly less frequency than implementation faults, but require about the same effort to find them and more effort to fix them.

In table 8, we compare interface and implementation faults with respect to their underlying causes. Underlying causes "other", "ambiguous requirements", "none given", "earlier incorrect fix" and "ambiguous design" tended to be the underlying causes more for implementation faults than for interface faults. Underlying causes "incomplete/omitted requirements", "incorrect modification" and "submitted under duress" tended to be the causes more for interface faults than for implementation faults.

We note that underlying causes that involved ambiguity tended to result more in implementation faults than in interface faults, while underlying causes involving incompleteness or omission of information tended to result more in interface faults than in implementation faults.

In table 9, we compare interface and implementation faults with respect to the means of prevention. Not surprisingly "formal requirements" and formal interface requirements" were more applicable to interface faults than to implementation faults. "Training", "expert person/documentation" and "guideline enforcement" were considered more applicable to implementation faults than to interface faults.

Table 8. Interface/Implementation Faults and Underlying Causes

	Interface 49%	Implementation 51%
1 none given	45.2%	54.8%
2 incomplete/omitted requirements	79.6%	20.4%
3 ambiguous requirements	44.5%	55.5%
4 incomplete/omitted design	50.8%	49.2%
5 ambiguous design	47.0%	53.0%
6 earlier incorrect fix	45.1%	54.9%
7 lack of knowledge	49.2%	50.8%
8 incorrect modification	54.5%	45.5%
9 submitted under duress	63.1%	36.9%
10 other	39.1%	60.1%

Table 9. Interface/Implementation Faults and Means of Prevention

	Interface 49%	Implementation 51%
1 formal requirements	64.8%	35.2%
2 requirements/design templates	51.5%	48.5%
3 formal interface specifications	73.6%	26.4%
4 training	36.4%	63.6%
5 application walk-troughs	48.0%	52.0%
6 expert person/documentation	44.3%	55.7%
7 design/code currency	46.7%	53.3%
8 guideline enforcement	33.1%	66.9%
9 better test planning	48.0%	52.0%
10 others	49.3%	50.7%

6 Conclusions

We have observed a large number of interesting facts about faults, the cost of finding and fixing them, their underlying causes and means of prevention. We offer the following general conclusions from these observations.

- The evolution of large, complex software systems involves a large overhead: approximately 51% of the MRs in the initial survey represented production faults, while 49% represented overhead faults(such as "duplicate" MRs, "no problem" MRs, and MRs on the system test environment).
- Interface faults were roughly 49% of the entire set of design and coding faults and were harder to fix than the implementation faults. Not surprisingly, formal requirements and formal interface specifications were suggested as significant means of preventing interface faults.
- Lack of information tended to dominate the underlying causes and knowledge intensive activities tended to dominate the means of prevention. Clearly, discovery (and rediscovery) are significant in the evolution of a very large real-time system.
- Relatively few problems would be solved by "better" programming languages (e.g., language pitfalls and race-conditions account for less than 8% of the faults). Technology that helps manage complexity and dependencies would

be much more useful (e.g., internal functionality, interface complexity, unexpected dependencies, low-level logic, and design/code complexity account for 60% of the faults).

The system reported here was developed and evolved using the current "best practice" techniques and tools with well-qualified practitioners. Because of this fact, we feel that this development is generalizable to other large-scale, real-time systems. With this in mind, we offer the following recommendations to improve the current "best practice".

– Obtain fault data throughout the entire development/evolution cycle (not just in the testing cycle) and use it monitor the progress of the evolution process [9].
– Incorporate the non-technological, people-intensive means of prevention into the current process. As our survey has shown, this will yield benefits for the majority of the faults reported here.
– Introduce facilities to increase the precision and completeness of requirements [10,22], architecture and design documents [17] and to manage complexity and dependencies [15]. This will yield benefits for those faults that were generally harder to fix and will help to detect the requirements, architecture and design problems earlier in the life-cycle.

Acknowledgements David Rosik contributed significantly to the general MR survey; Steve Bruun produced the cross-tabulated statistical analysis for the design/code survey and contributed, along with Carolyn Larson, Julie Federico, H. C. Wei and Tony Lenard, to the analysis of the design/code survey; Clive Loader increased our understanding of the chi-square analysis; and Larry Votta helped refine and improve our presentation. We especially thank Marjory P. Yuhas and Lew G. Anderson for their unflagging support of this work. And finally, we thank all those that participated in the survey.

References

1. Basili, Victor R., Hutchens, David H.: An Empirical Study of a Syntactic Complexity Family IEEE Transactions on Software Engineering SE-9:6 (November 1983) 664-672
2. Basili, Victor R., Perricone, Barry T.: Software Errors and Complexity: an Empirical Investigation. Communications of the ACM 27:1 (January 1984) 42-52
3. Basili, Victor R., Weiss, David M.: A Methodology for Collecting Valid Software Engineering Data. IEEE Transactions on Software Engineering SE-10:6 (November 1984) 728-738
4. Boehm, Barry W.: Software Engineering Economics. Englewood Cliffs: Prentice-Hall, 1981
5. Bowen John B.: Standard Error Classification to Support Software Reliability Assessment. AFIPS Conference Proceedings, 1980 National Computer Conference (1980) 697-705

6. Curtis, Bill, Krasner, Herb, Iscoe, Neil: A Field Study of the Software Design Process for Large Systems. Communications of the ACM 31:11 (November 1988) 1268-1287
7. Endres, Albert: An Analysis of Errors and Their Causes in System Programs. IEEE Transactions on Software Engineering SE-1:2 (June 1975) 140-149
8. Glass, Robert L.: Persistent Software Errors. IEEE Transactions on Software Engineering SE-7:2 (March 1981) 162-168
9. Humphrey, Watts S.: Managing the Software Process. Reading, Mass: Addison-Wesley, 1989.
10. Kelly, Van E., Nonnenmann, Uwe: Inferring Formal Software Specifications from Episodic Descriptions. Proceedings of AAAI 87. Sixth National Conference on Artificial Intelligence (13-17 July 1987) Seattle WA, 127-132
11. Lehman, M. M., Belady, L. A.: Program Evolution. Processes of Software Change. London: Academic Press, 1985
12. Ostrand, Thomas J., Weyuker, Elaine J.: Collecting and Categorizing Software Error Data in an Industrial Environment. The Journal of Systems and Software 4 (1984) 289-300
13. Perry, Dewayne E., Evangelist, W. Michael: An Empirical Study of Software Interface Errors. Proceedings of the International Symposium on New Directions in Computing, IEEE Computer Society (August 1985) Trondheim, Norway, 32-38
14. Perry, Dewayne E., Evangelist, W. Michael: An Empirical Study of Software Interface Faults —An Update. Proceedings of the Twentieth Annual Hawaii International Conference on Systems Sciences (January 1987) Volume II 113-126
15. Perry, Dewayne E.: The Inscape Environment. Proceedings of the 11th International Conference on Software Engineering, (15-18 May 1989) Pittsburgh PA, 2-12
16. Perry, Dewayne E.: Industrial Strength Software Development Environments. Proceedings of IFIPS Congress '89 —11th World Computer Congress (August 28 - September 1, 1989) San Francisco CA
17. Perry, Dewayne E., Wolf, Alexander L.: Foundations for the Study of Software Architecture. ACM SIGSOFT Software Engineering Notes 17:4 (October 1992) 40-52
18. Rowland, B. R., Anderson, R. E., McCabe, P. S.: The 3B20D Processor & DMERT Operating System: Software Development System. The Bell System Technical Journal 62:1 part 2 (January 1983) 275-290.
19. Schneidewind, N. F., Hoffman, Heinz-Michael: An Experiment in Software Error Data Collection and Analysis", IEEE Transactions on Software Engineering SE-5:3 (May 1979) 276-286
20. Siegel, Sidney, and Castellan, Jr., N. John: Nonparametric Statistics for the Behavioral Sciences. Second Edition. New York: McGraw-Hill, 1988
21. Thayer, Thomas A., Lipow, Myron, Nelson, Eldred C.: Software Reliability - A Study of Large Project Reality. TRW Series of Software Technology, Volume 2. North-Holland, 1978.
22. Zave, Pamela, Jackson, Daniel: Practical Specification Techniques for Control-Oriented Systems. Proceedings of IFIPS Congress '89 —11th World Computer Congress (August 28 - September 1, 1989) San Francisco CA

This article was processed using the LaTeX macro package with LLNCS style