CHAPTER TWENTY-FIVE

Where Do Most Software Flaws Come From?

Dewayne Perry

The holy grail of software development management is "cheaper, faster, and better." Unfortunately, a lot of poor management decisions are made in pursuit of this grail. While "cheaper and faster" are often very important, "better" clearly is the most important in a wide variety of software systems where reliability or safety is of paramount importance.

There are a variety of different ways in which a product can be made better, ranging from more clearly understanding customer needs to minimizing faults in the software system. It is the latter that is the focus of this chapter. Only by understanding the mistakes we make can we determine what remedies need to be applied to improve either the products or the processes. Monitoring faults is a relatively simple matter, either as they are found or in project retrospectives (often referred to as "project post-mortems").

A fundamental aspect in minimizing faults in software systems is the managing of complexity, the most critical of essential characteristics of software systems [Brooks 1995]. One of the most useful techniques in managing that complexity is that of separating the interfaces of components from their implementations. It is because of this critical technique that the difference between interface and implementation faults is an important distinction that is addressed in this chapter.

AU: THIS WAS BLANK, FILL IN

On the one hand, it is frustrating that so few studies of software faults have been published to guide researchers in finding ways of detecting, ameliorating, or preventing these faults from happening. On the other hand, it is not at all surprising that projects are reluctant to make such sensitive data public because of internal politics or external competitiveness in software-intensive businesses.

Despite this paucity of studies and the reluctance of companies to make data available, there is a set of landmark studies about software faults that provide useful foundations for product and process improvements. Endres [Endres 1975], Schneidewind and Hoffmann [Schneidewind and Hoffmann 1979], and Glass [Glass 1981] reported on various fault analyses of software development. A weakness in their work is that they do not delineate interface faults as a specific category.

Thayer, Lipow, and Nelson [Thayer et al. 1978] and Bowen [Bowen 1980] provide extensive categorization of faults, but with a relatively narrow view of interface faults. Basili and Perricone [Basili and Perricone 1984] 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 note, however, that none of these studies address the kinds of problems that arise in very large-scale software developments, nor do they address the evolutionary phase of developments. Perry and Evangelist [Perry and Evangelist 1985], [Perry and Evangelist 1987] were the first to address fault studies in the evolution of a large real-time system. An extremely important factor in this study is the fact that interface faults were by far the overwhelming and dominant faults (68% of the faults). An important question that was left unanswered was whether these interface faults were the easy or the hard ones to find and fix.

The distinction between an evolutionary software system release and an initial development release is a critical one. In the latter case, the design and implementation choices are much less constrained than in evolutionary development. In the former, you have to make changes to an existing system, and so the choices are far more constrained and there are many more difficulties in understanding the implications of changes. As the evolutionary development part of a system's life cycle is far greater than its initial development, so too are studies of the faults in that evolutionary part much more important.

In this study, we take a detailed look at one specific release of one ultra-reliable, ultra-large-scale, real-time system rather than a more superficial look at several more moderately sized systems in several domains. The advantage of this approach is that we gain a deeper understanding of the system and its problems. The disadvantage is that we are less able to generalize our results compared to the latter course. This type of trade-off is often encountered in empirical studies.

As we will see, however, this deeper look provides us with a number of practical and useful insights. For example, it is commonly accepted wisdom that "once a bug is found, it is easy to fix." Unfortunately, our data contradicts this "common wisdom." Or, in the case of the unanswered question about interface faults, our data supports our original but unsubstantiated intuition that interface faults are harder to fix than implementation faults.

Context of the Study

The system discussed in this chapter is a very large-scale (that is, a million lines or more), distributed, real-time system written in the C programming language (with additional domain-specific languages as needed) in a Unix-based, multiple machine, multiple location environment.

The organizational structure is typical with respect to 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 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 3 to 15 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 phase are subjected both to formal code reviews by three to five reviewers and to low-level unit testing. As components are 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, new functionality, etc.) are generally uniform across releases. It is because of these two facts that we consider this study to provide a representative sample in the life of the project.

Faults discovered during testing phases are reported and monitored by a modification request (MR) tracking system (such as, for example, CMS [Rowland et al. 1983]). Access to source files for modification is possible only through the tracking system. Thus all change activity (whether fixing faults, adding new functionality, or improving existing functionality—that is, whether they are corrective, adaptive, or perfective changes) 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.

The goal of this study was to gain insight into the current process of system evolution by concentrating on one release of a particular system. The approach we used is that of surveying, by means of a prepared questionnaire, those developers who "owned" the fault MR at the time it was closed, surveying first the complete set of faults and then concentrating on the largest set of faults in more depth. This survey was the first of its type, although there have been some smaller studies using random selections. The CMS MR database was used to determine the initial set of fault MRs to survey and the developers who were responsible for closing those fault MRs. The survey identifying the fault MR was then sent to the identified developer to complete.

For a variety of reasons (schedule pressure among them), there were significant constraints placed on the study by project management: first, the study had to be completely non-intrusive; second, it had to be strictly voluntary; and third, it had to be completely anonymous. We will see in the later discussion about validity issues that these mandates were the source of some study weaknesses.

It is with this background that we present our surveys, analyses, and results.

Phase 1: Overall Survey

There were three specific purposes in the original overall survey:

- To determine, generally, what kinds of problems were found (which we report here), as well as, specifically, what kinds of application-specific problems arose during the preparation of this release (which we do not report, because of their lack of generality)
- To determine how the problem was found (that is, in which testing phase)
- To determine when the problem was found

One of the problems encountered in any empirical survey study is ensuring that the survey is in the "language" of those being surveyed. By this "language" we mean both the company- or project-specific jargon that is used but also the process that is being used. You want the developer surveyed to clearly understand what is being asked in terms of his own context. Failing to understand this results in questions about the validity of the survey results. To this end, we used developers to help us design the survey, and we used the project jargon and process to provide a familiar context.

Summary of Questionnaire

The first phase of the survey questionnaire had two main components: the determination of the category 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 fault MRs were initiated and when they were closed) was retrieved from the MR tracking system database.

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 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 size and complexity of the system, while at the same time increasing its load and stress.

Summary of the Data

Table 25-1 summarizes the fault MRs by fault category. The fault 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 fault MRs. Given that the distinction between a design fault and a coding fault required a "judgment call" on the part of the respondent, we decided to merge the results of those two categories into one: design/coding faults account for 28.8% of the MRs. However, in the process structure used in the project, the distinction between requirements and design/coding is much clearer. Requirements specifications are produced by systems engineers, whereas the design and coding are done by developers.

TABLE 25-1. Summary of faults

MR category	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%

The next most significant subset of MRs were those that concern testing (the testing environment and testing categories)—24.8% of the MRs. On the one hand, 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. On the other hand, this general problem is clearly one that needs to be addressed by further study.

"Duplicate" and "no problem" MRs account for another significant subset of the data—29.8%. Historically, these have been considered to be part of the overhead. Certainly the "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. The "no problem" MRs, however, are in large part due to the lack of understanding that comes from informal and out-of-date documentation. Obviously, measures taken to reduce these kinds of problems will have beneficial effects on other categories as well. In either case, reduction of administrative overhead will improve the cost effectiveness of the project.

"Previous" MRs indicate the level of difficulty in finding some of the faults in a large, real-time system. These problems may have been impossible to find in the previous releases and have only now been exposed because of changes in the use of the system.

Figure 25-1 charts the requirements, design, and coding MRs by testing phase. We have focused on this early part of the software development process because that is where the most MRs occurred and, accordingly, where closer attention should yield the most results.

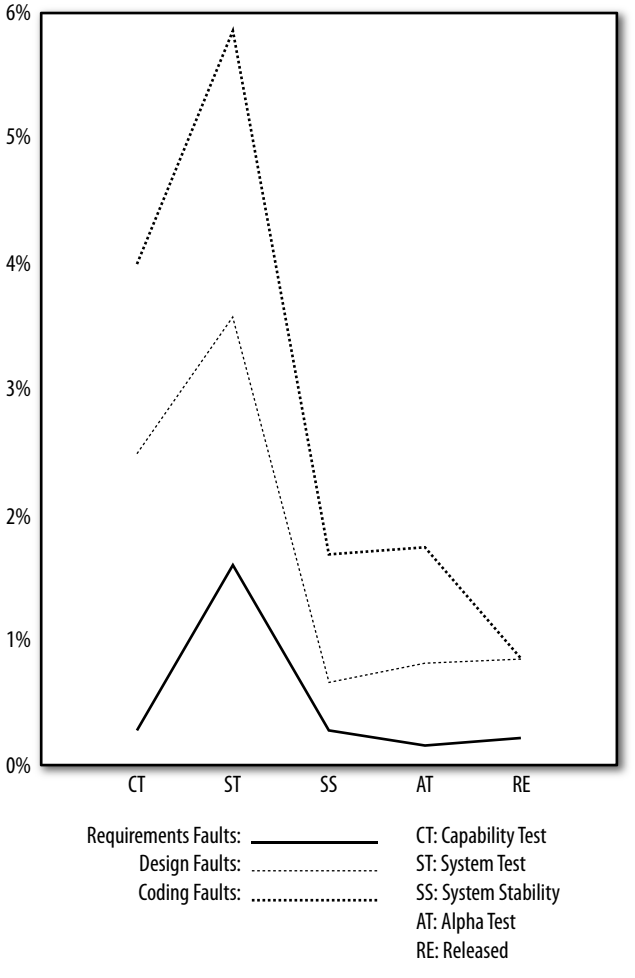

FIGURE 25-1. Fault categories found by phase

Please note that the percentages used in Figure 25-1 and Figure 25-2 are the percentages of those faults relative to all the faults, not the percentages relative to just the charted faults. The phases in Figure 25-1 appear as sequential when in actual fact (as is almost always the case in software systems' development) there is a lot of parallelism, with phases overlapping significantly. With hundreds of software engineers developing hundreds of features concurrently, the actual project life cycle is nothing like the sequential waterfall model, even though software development proceeds through a set of, often iterative, phases. That is the reason for Figure 25-2, which shows the same data relative to a fixed timeline.

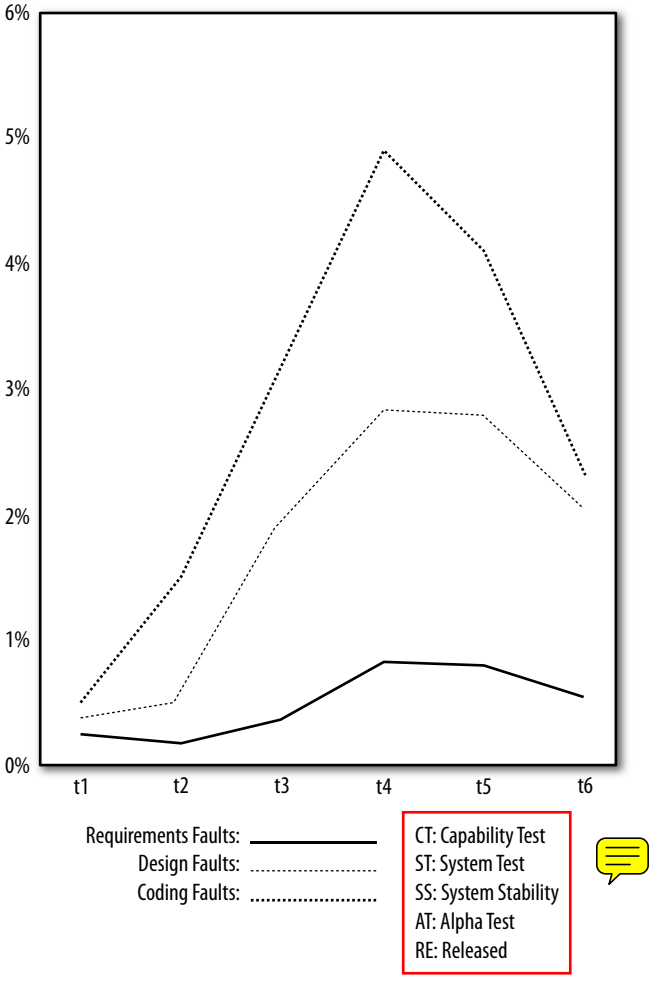

FIGURE 25-2. Fault categories found over time

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. This is not particularly surprising, since that is when we were looking the hardest for, and expecting the most, faults. System test is where the various components are first integrated and is the most likely place to encounter the mismatches and misassumptions among components. So while all testing is looking for faults, system test is where we look the hardest. Capability testing is akin to unit testing in an appropriate context—internal and initial interaction faults are looked for. The capability test context limits the kinds of faults that can be exposed.

Second, all testing phases found MRs of each fault category. It is also not surprising that coding faults are found over the entire set of testing phases. One obvious reason for this phenomena is that changes are continually made to correct the faults that are found in the various earlier testing phases. Moreover, while it is disturbing to note that both design and requirements faults continue to be found throughout the entire testing process, we feel that this is due to the lack of precision and completeness in requirements and design documentation and is a general problem in the current state-of-practice rather than a project-specific problem.

The time values in Figure 25-2 are fixed intervals. From the shape of the data, it is clear that System Testing overlaps interval t4. It is unfortunate that we have only the calendar data (that is, the time boundaries on the MRs), as a correlation with effort data [Musa et al. 1987] (that is, the actual amount of time spent in that time period) would be extremely valuable.

For the requirements, design, and coding fault categories over time, Figure 25-2 shows that all fault types peaked at time t4 and held through time t5, except for the coding faults, which decreased.

Summary of the Phase 1 Study

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

- Implementation, testing, and administrative overhead faults occurred in roughly equal proportions
- Requirements problems, while not overwhelmingly numerous, are still significant (especially since the majority were found late in the testing process)
- All types of faults continued to be found in all the testing phases
- The most faults were found when the level of testing effort was the highest (that is, at system test)
- The majority of faults were found late in the testing cycle

These observations are limited by the fact that the tracking of fault MRs is primarily a testing activity. It would be extremely useful to observe the kinds and frequencies of faults that exist 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.

Phase 2: Design/Code Fault Survey

As a result of the general survey, we decided to survey 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
- Determine how the faults might have been prevented
- Compare the difficulty in finding and fixing interface and implementation faults

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 [Boehm 1981](see also Chapter 10 by Barry Boehm). Thus, gaining insight into these problems will yield significant and cost beneficial results.

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

The Questionnaire

The respondents were asked to indicate the difficulty of finding and fixing the problem, determine the actual and underlying causes, indicate the best means of either preventing or avoiding the problem, and give their level of confidence in their responses. It should be kept in mind that the people surveyed were those who owned the MR at the time it was closed (i.e., completed).

- For each MR, rank it 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.
- For each MR, how much time was needed to design and code the fix, and document and test it. (Note that what would be an easy fix in a single-programmer system takes considerably more time in a large, multiperson project with a complex laboratory test environment.)
    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

- For each MR, consider the following 22 possible types and select the one that most closely applies to the immediate cause (that is, the fault type).
    1. *Language pitfalls*—for example, pointer problems, or the use of "=" instead of "= =".
    2. *Protocol*—violated rules about interprocess communication.
    3. *Low-level logic*—for example, loop termination problems, pointer initialization, etc.
    4. *CMS complexity*—for example, due to software 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 that were not *used* correctly.
    8. *Primitives unsupported*—the design or code depended on primitives that were not adequately developed (that is, the primitives do 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*—for example, 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.

- 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.
- For this fault, consider possible ways to prevent or avoid it, and select the most useful or appropriate choice for preventing, avoiding, or detecting 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 requested of the respondents were: *very high, high, moderate, low*, and *very low*. We discarded the small number of responses that had a confidence level of either low or very low.

Statistical Analysis

Out of all the questionnaires, 68% were returned. Of those, we dropped the responses that were either low or very low in confidence (6%). The remainder of the questionnaires were then subjected to Chi-Square analysis [Siegel et al. 1988] to test for independence (and for interdependence) of various paired sets of data. In Chi-Square analysis, the lower the total chi-square value, the more independent the two sets of data; the higher the value, the more interdependent the two sets of data. The p-value indicates the significance of the analysis: the lower the number, the less likely the relationships are due to chance. In Table 25-2, the prevention data and the find data are the most independent (the total chi-square is the lowest), and that lack of relationship is significant (the p-value is less than the standard .05 and indicates that the odds are less than 1 in 20 that the relationship happened by chance). The fault-cause, fault-prevention, and cause-prevention pairs are the most interdependent, as their total chi-square values are the largest three of the entire set and the significance of these relationships is very high (the odds are less than 1 in 10,000 of being by chance).

The fact that the relationships between the faults and their underlying causes, faults and means of prevention, and means of prevention and the underlying causes are the most significantly interdependent is a good thing: 1) faults should be strongly related to their underlying causes, and 2) both faults and their underlying causes should be strongly related to their means of prevention. This indicates that the respondents were consistent in their responses and the data aligns with what one would logically expect.

TABLE 25-2. 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

Variables	Degrees of freedom	Total Chi-Square	p
Prevention, Fix	27	97.886	.0001
Fault, Prevention	189	492.826	.0001
Cause, Prevention	81	641.417	.0001

Finding and fixing faults

Table 25-3 provides a cross-tabulation of the difficulty in finding and fixing the design and coding faults. Of these faults, 78% took five days or less to fix. In general, the easier-to-find faults were easier to fix; the more difficult-to-find faults were more difficult to fix.

TABLE 25-3. Find versus fix comparison

Find/fix		<1 day	1-5 days	6-30 days	>30 days
		30.1%	**48.8%**	**18.0%**	**3.6%**
easy	**67.5%**	23.7%	32.1%	10.0%	1.7%
moderate	**23.4%**	4.2%	12.5%	5.6%	1.1%
difficult	**7.7%**	1.7%	3.4%	2.1%	.5%
very difficult	**1.4%**	.5%	.3%	.3%	.3%

One of the interesting things about Chi-Square analysis is that it is based on the difference between expected and observed values of the paired data. The expected value in this case is the product of the observed find value and the observed fix value. If the two sets of data are independent of each other, the expected percentages will match or be very close to the observed percentages; otherwise, the two sets of data are not independent.

The first row of data is the observed percentages of how long it took to fix the MR; the first column is the observed percentages of how hard it was to find/duplicate the problem. The expected value of easy to find and fixable in a day or less is 67.1% x 30.1% = 20.2%, whereas the actually observed value of 23.7% is 17% more than that expected value.

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 (expected: 12%) that took 6 to 30 days to fix (observed: 10%).

Although the coordinates of the effort to find and fix the faults are non-comparable, we note that the following relationship is suggestive. Collapsing the previous table yields an interesting insight in Table 25-4 that 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.

TABLE 25-4. Summary of find/fix

Find/fix effort	≤ 5 days	≥ 6 days
easy/moderate	72.5%	18.4%
difficult/very difficult	5.9%	3.2%

Faults

Table 25-5 shows the fault types of the MRs as ordered by their frequency in the survey independent of any other factors. For the sake of brevity in the subsequent tables, we use the fault type number to represent the fault types.

The first five fault types account for 60% of the 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 fault types are leading faults is consistent with the nature of the evolution of the system. Adding significant amounts of 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 relatively insignificant is probably due to the fact that this is not an early release of the system, and performance was always a significant concern of code inspections.

Fault Frequency Adjusted by Effort

There are two interesting relationships to consider in the ordering of the various faults: the effect that the difficulty in *finding* the faults has on the ordering and the effect that the difficulty of *fixing* the faults has on the ordering. The purpose of weighting is to provide an adjustment to the observed frequency by how easy or hard the faults are to find or to fix. From the standpoint of "getting the most bang for the buck," the frequency of a fault is a good prima facie indicator of the importance of one fault relative to another. Table 25-5 shows the fault types ordered by frequency.

TABLE 25-5. Fault types ordered by frequency

Fault type	Observed %	Fault type description
5	25.0%	internal functionality
10	11.4%	interface complexity
20	8.0%	unexpected dependencies
3	7.9%	low-level logic
11	7.7%	design/code complexity
22	5.8%	other
9	4.9%	change coordination
21	4.4%	concurrent work
13	4.3%	race conditions
6	3.6%	external functionality
1	3.5%	language pitfalls
12	3.3%	error handling
7	2.4%	primitives misused
17	2.1%	dynamic data use
15	1.5%	resource allocation
18	1.0%	static data design
14	.9%	performance
19	.7%	unknown interactions
8	.6%	primitives unsupported
2	.4%	protocol
4	.3%	CMS complexity
16	.3%	dynamic data design

Table 25-6 is an attempt to capture the weighted difficulty of finding the various faults. The weighting is done by multiplying the proportion of observed values for each fault with multiplicative weights of 1, 2, 3, and 4 for each find category, respectively, and summing the results.

Obviously it would have been better to have had some duration assigned to the effort to find faults and then correlated the weighting with those durations, as we do subsequently in weighting by effort to fix faults. The weights used are intended to be suggestive, not definitive.

We experimented with several different weightings, and the results were pretty much the same. Thus we used the simplest approach.

Better yet would have been effort data associated with each MR that could be used to get a more realistic picture of actual difficulty. But this type of data is seldom available, and an approximation is needed instead.

For example, if a fault was easy to find in 66% of the cases, moderate in 23%, difficult in 11%, and very difficult in 0%, the weight is 145 = (66 * 1) + (23 * 2) + (11 * 3) + (0 * 4).
Table 25-6 shows the fault types weighted by difficulty to find, from easiest to most difficult.

TABLE 25-6. Determining the find weighting

Fault type	Find proportion e/m/d/vd	Weight	Fault type description
4	100/0/0/0	100	CMS complexity
18	100/0/0/0	100	static data design
7	88/8/4/0	120	primitives misused
2	75/25/0/0	125	protocol
20	78/16/5/1	129	unexpected dependencies
21	70/23/2/4	130	concurrent work
3	73/22/5/0	132	low-level logic
22	82/12/2/5	132	other
5	74/19/6/1	134	internal functionality
6	67/31/3/0	139	external functionality
1	68/26/2/2	141	language pitfalls
10	66/23/11/0	145	interface complexity
9	65/20/12/2	149	change coordination
8	67/17/17/0	152	primitives unsupported
19	88/8/4/0	157	unknown interactions
16	67/0/33/0	157	dynamic data design
17	52/38/10/0	158	dynamic data use
15	47/47/7/0	162	resource allocation
12	55/30/12/3	163	error handling
11	55/29/16/1	165	code complexity
14	56/11/11/22	199	performance
13	12/67/21/0	209	race conditions

Typically, performance faults and race conditions are very difficult to isolate and reproduce. We would expect that "code complexity" and "error handling" faults also would be difficult to find and reproduce. Not surprisingly, "language pitfalls" and "interface complexity" are reasonably hard to detect.

In the Chi-Square analysis, "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 population was larger.

If we weight the proportions by multiplying the number of occurrences of each fault by its weight from Table 25-5 and dividing by the total weighted number of occurrences, we get only a slight change in the ordering of the faults, with "internal functionality," "code complexity," and "race conditions" (faults 5, 11, and 13) changing slightly more than the rest of the faults.

Table 25-7 represents the results of weighting the difficulty of fixing the various faults by factoring in the actual time needed to fix the faults. The multiplicative scheme uses the values 1, 3, 15, and 30 for the four average times in fixing a fault. The calculations are performed as in the example of weighting the difficulty of finding the faults.

The weighting according to the difficulty in fixing the fault causes some interesting shifts in the ordering. "Language pitfalls," "low-level logic," and "internal functionality" (faults 1, 3, and 5) drop significantly in their relative importance. This coincides with one's intuition about these kinds of faults. "Design/code complexity," "resource allocation," and "unexpected dependencies" (faults 11, 15, and 20) rise significantly in their relative importance; "interface complexity," "race conditions," and "performance" (faults 10, 13, 14) also rise, but not significantly so.

TABLE 25-7. Determining the fix weighting

Fault type	Proportion e/m/d/vd	Weight	Fault type description
16	67/33/0/0	166	dynamic data design
4	67/33/0/0	166	CMS complexity
8	50/50/0/0	200	primitives unsupported
18	50/50/0/0	200	static data design
1	63/31/6/0	244	language pitfalls
3	59/37/3/1	245	low-level logic
2	25/75/0/0	250	protocol
17	38/48/14/0	392	dynamic data use
9	37/49/14/0	394	change coordination
5	27/59/14/0	414	internal functionality

Fault type	Proportion e/m/d/vd	Weight	Fault type description
22	40/43/12/5	496	other
7	46/37/8/8	497	primitives misused
10	17/57/26/1	608	interface complexity
21	25/43/30/2	661	concurrent work
6	22/50/22/6	682	external functionality
13	16/56/21/7	709	race conditions
12	21/52/18/9	717	error handling
19	29/43/14/14	785	unknown interactions
20	24/39/33/5	786	unexpected dependencies
11	22/39/27/12	904	design/code complexity
14	11/22/44/22	1397	performance
15	0/47/27/27	1356	resource allocation

Table 25-8 shows the top fix-weighted faults. According to our weighting schemes, these four faults account for 55.2% of the effort expended to fix all the faults and 51% of the effort to find them, but represent 52.1% of the faults by frequency count. Collectively, they are somewhat harder to fix than rest of the faults and slightly easier to find. We again note that although the two scales are not strictly comparable, the comparison is an interesting one nonetheless.

TABLE 25-8. Faults weighted by fix difficulty

Fault type	Weighted %	Brief description
5	18.7%	internal functionality
10	12.6%	interface complexity
11	12.6%	code complexity
20	11.3%	unexpected dependencies

In the Chi-Square analysis, "language pitfalls" and "low-level logic" took fewer days to fix than expected. "Interface complexity" and "internal functionality" took 1 to 6 days to fix more often than expected, and "design/code complexity" and "unexpected dependencies" took longer to fix (that is, 6 to over 30 days) than expected. These deviations reinforce our weighted assessment of the effort to fix the faults.

Underlying causes

In Table 25-9, we show the underlying causes of the MRs as ordered by their frequency in the survey, independent of any other factors.

TABLE 25-9. Underlying causes of faults

Underlying causes	Observed %	Brief description
4	25.2%	incomplete/omitted design
1	20.5%	none given
7	17.8%	lack of knowledge
5	9.8%	ambiguous design
6	7.3%	earlier incorrect fix
9	6.8%	submitted under duress
2	5.4%	incomplete/omitted requirements
10	4.1%	other
3	2.0%	ambiguous requirements
8	1.1%	incorrect modification

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 faults in the "none given" underlying cause category in Table 25-12 below). 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 (3.4%—or 16% of the faults in the "none given" underlying cause category in Table 25-12). On the other hand, we were surprised that no cause was given for a substantial part of the "internal functionality" faults (3.3%—or 16% of the faults in the "none given" category in Table 25-12). One would expect there to be some underlying cause for that particular fault.

Table 25-10 shows the relative difficulty in finding the faults associated with the underlying causes. The resulting ordering is particularly nonintuitive: the MRs with no underlying cause are the second most difficult to find; those submitted under duress are the most difficult to find.

TABLE 25-10. Weighting of the underlying causes by find effort

Underlying causes	Proportion	Weight	Brief description
8	91/9/0/0	109	incorrect modification
7	74/18/7/1	135	lack of knowledge
3	60/40/0/0	140	ambiguous requirements
5	66/27/7/0	141	ambiguous design
2	70/17/13/0	143	incomplete/omitted requirements
4	68/25/7/1	143	incomplete/omitted design
6	73/12/10/5	147	earlier incorrect fix
10	76/12/0/12	148	other
1	63/25/11/1	150	none given
9	50/46/4/0	158	submitted under duress

In the Chi-Square analysis of finding underlying causes, faults caused by "lack of knowledge" tended to be easier to find than expected, whereas faults caused by "submitted under duress" tended to be moderately hard to find more often than expected. This latter finding is interesting, as we know very little about faults "submitted under duress."

In Table 25-11, we weight the underlying causes by the effort to fix the faults represented by the underlying causes. This 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; and "unclear design" and "other" decreased less significantly. However, the relative ordering of the various underlying causes is unchanged.

TABLE 25-11. Weighting of the underlying causes by fix effort

Underlying causes	Proportion	Weight	Brief description
10	37/42/12/10	340	other
1	43/43/12/2	412	none given
5	29/55/14/2	464	ambiguous design
7	30/50/17/3	525	lack of knowledge
6	34/45/17/4	544	earlier incorrect fix
9	18/57/25/0	564	submitted under duress
8	18/55/27/0	588	incorrect modification
4	23/50/22/5	653	incomplete/omitted design

Underlying causes	Proportion	Weight	Brief description
2	26/44/24/6	698	incomplete/omitted requirements
3	25/30/24/6	940	ambiguous requirements

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

In the Chi-Square analysis of fixing underlying causes, faults caused by "none given" tended to take less time to fix than expected, whereas faults caused by "incomplete/omitted design" and "submitted under duress" tended to take more time to fix than expected.

In Table 25-12, we present the cross-tabulation of faults and their underlying causes. Faults are represented by the rows, underlying causes by the columns. The numbers in the matrix are the percentages of the total population of faults. Thus, 1.5% of the total faults were fault 1 with the underlying cause 1. The expected number of faults for fault 1 and underlying cause 1 can be computed by multiplying the total faults for each of those categories: 20.5% * 3.5% = .7%. In this example, the actual number of faults was higher than expected.

TABLE 25-12. Cross-tabulating fault types and underlying causes

		1	2	3	4	5	6	7	8	9	10
		20.5%	**5.4%**	**2.0%**	**25.2%**	**9.8%**	**7.3%**	**17.8%**	**1.1%**	**6.8%**	**4.1%**
1 language pitfalls	**3.5%**	1.5	.0	.0	.2	.1	.2	.8	.1	.5	.1
2 protocol	**.4%**	.0	.0	.1	.2	.0	.0	.1	.0	.0	.0
3 low-level logic	**7.9%**	3.7	.3	.1	.6	.3	1.2	.7	.0	.6	.4
4 CMS complexity	**.3%**	.1	.0	.0	.0	.0	.1	.1	.0	.0	.0
5 internal functionality	**25.0%**	3.3	1.3	.6	7.7	2.8	2.0	5.2	.3	1.2	.6
6 external functionality	**3.6%**	.7	.3	.1	.4	.5	.6	.7	.0	.3	.0
7 primitives misused	**2.4%**	.4	.0	.0	.5	.0	.1	.8	.0	.0	.6
8 primitives unsupported	**.6%**	.0	.2	.0	.1	.0	.1	.1	.0	.1	.0
9 change coordination	**4.9%**	1.1	.0	.0	.8	1.0	.6	.8	.1	.3	.2

		1	2	3	4	5	6	7	8	9	10
		20.5%	**5.4%**	**2.0%**	**25.2%**	**9.8%**	**7.3%**	**17.8%**	**1.1%**	**6.8%**	**4.1%**
10 interface complexity	**11.4%**	2.1	.6	.2	4.1	1.4	1.1	1.4	.2	.0	.3
11 design/ code complexity	**7.7%**	1.3	.0	.3	3.0	1.6	.2	1.0	.0	.0	.3
12 error handling	**3.3%**	.9	.3	.0	.8	.0	.1	.7	.0	.4	.1
13 race conditions	**4.3%**	1.4	.2	.0	1.3	.5	.1	.3	.0	.4	.1
14 performance	**.9%**	.2	.0	.1	.2	.0	.0	.3	.0	.0	.1
15 resource allocation	**1.5%**	.5	.0	.0	.3	.1	.0	.4	.1	.0	.1
16 dynamic data design	**.3%**	.0	.0	.0	.1	.0	.0	.1	.0	.1	.0
17 dynamic data use	**2.1%**	.7	.1	.0	.2	.1	.0	.6	.0	.4	.0
18 static data design	**1.0%**	.3	.1	.1	.2	.1	.0	.1	.0	.1	.0
19 unknown interactions	**.7%**	.0	.1	.1	.0	.2	.0	.2	.0	.1	.0
20 unexpected dependencies	**8.0%**	.5	.8	.3	2.7	.5	.1	1.4	.0	1.7	.0
21 concurrent work	**4.4%**	.6	.3	.0	1.2	.2	.4	.9	.2	.4	.2
22 other	**5.8%**	1.2	.8	.0	.6	.4	.4	1.1	.1	.2	1.0

For the sake of brevity, we consider only the most frequently occurring faults and their major underlying causes. "Incomplete/omitted design" (cause 4) is the primary underlying cause in all of these major faults. "Ambiguous design" (cause 5), "lack of knowledge" (cause 7), and "none given" (cause 1) were also significant contributors to the presence of these faults.

internal functionality (fault 5)

"incomplete/omitted design" (cause 4) was felt to have been the cause of 31% (that is, 7.7% / 25%) of the occurrences of this fault, a percentage higher than expected; "lack of knowledge" (cause 7) was thought to have caused 21% of the occurrences of this fault, higher than expected; and "none given" was listed as the third underlying cause, representing 13% of the occurrences.

interface complexity (fault 10)

Again, "incomplete/omitted design" was seen to be the primary cause in the occurrence of this fault (36%), higher than expected; "lack of knowledge" and "ambiguous design" were seen as the second and third primary causes of this fault (13% and 12%, respectively).

unexpected dependencies (fault 20)

Not surprisingly, "incomplete/omitted design" was felt to have been the primary cause of this fault (in 34% of the cases); "submitted under duress" (cause 9) contributed to 21% of the occurrences, a percentage higher than expected; and "lack of knowledge" was the tertiary cause of this fault, representing 18% of the occurrences.

design/code complexity (fault 11)

Again, "incomplete/omitted design" was felt to have been the primary cause in 39% of the occurrences of this fault, a percentage higher than expected; "ambiguous design" was the second most frequent underlying cause of this fault, causing 21% of the faults (also a higher percentage than expected); and "none given" was listed as the third underlying cause, representing 17% of the occurrences.

Again, for the sake of brevity, we consider only the most frequently occurring underlying causes and the faults to which they were most applicable.

incomplete/omitted design (cause 4)

As we noted previously, "internal functionality," "interface complexity," "code/design complexity," and "unexpected dependencies" were the major applicable faults (31%, 12%, 12%, and 11%, respectively), with the first three occurring with higher than expected frequency.

none given (cause 1)

"low-level logic" (fault 3) was the leading fault, representing 18% of the occurrences (a percentage higher than expected); "internal functionality" (fault 5) was the second major fault, representing 16% of the occurrences (a percentage lower than expected); "interface complexity" (fault 10) was the third leading fault, representing 10% of the occurrences; and "language pitfalls" was the fourth leading fault, representing 8% of the occurrences (a percentage higher than expected).

lack of knowledge (cause 7)

"internal functionality" was the leading fault, representing 29% of the occurrences (a percentage higher than expected); "interface complexity" was next with 8% of the

occurrences (a percentage lower than expected); "unexpected dependencies" was third with 8% of the occurrences; and "other" (fault 22) was the fourth with 6%.

ambiguous design (cause 5)

"internal functionality" represented 29% of the occurrences; "code/design complexity" (fault 11) was second fault, representing 16% of the occurrences (a percentage higher than expected); "interface complexity" was third with 14%; and "change coordination" (fault 9) was fourth, representing 10% of the occurrences (a percentage higher than expected).

Means of prevention

Table 25-13 shows the means of prevention of the MRs, as ordered by their occurrence independent of any other factors. We note that the means selected may well reflect a particular approach of the responder in selecting one means over another (for example, see the discussion later in this section about formal versus informal means of prevention).

TABLE 25-13. Means of error prevention

Means of prevention	Observed %	Brief description
5	24.5%	application walk-throughs
6	15.7%	expert person/documentation
8	13.3%	guideline enforcement
2	10.0%	requirements/design templates
9	9.9%	better test planning
1	8.8%	formal requirements
3	7.2%	formal interface specifications
10	6.9%	other
4	2.2%	training
7	1.5%	design/code currency

It is interesting to note that the application-specific means of prevention ("application walk-throughs") is considered the most effective. This selection of application walk-throughs as the most useful means of error prevention appears to confirm the observation of Curtis, Krasner, and Iscoe [Curtis et al. 1988] that a thin spread of application knowledge is 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 nontechnical solution to providing the

information that may be supplied by formal representations (and which provide a more technical solution with perhaps higher attendant 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 Table 25-13, with the exception of "requirements/design templates," which seems to apply to the easier-to-find faults, and "guideline enforcement," which seems to apply more to the harder-to-find faults.

In the Chi-Square analysis, the relationship between finding faults and preventing them is the most independent of the relationships, reported here with p=.041. "Application walk-throughs" applied to faults that were marginally easier to find than expected, whereas "guideline enforcement" applied to faults that were less easy to find than expected.

In Table 25-14, the means of prevention is weighted by the effort to fix the associated faults.

TABLE 25-14. Means of prevention weighted by fix effort

Prevention	Proportion	Weight	Brief description
8	38/52/7/3	389	guideline enforcement
9	35/52/12/1	401	better test planning
7	40/40/20/0	460	design/code currency
5	33/50/17/1	468	application walk-throughs
10	49/36/6/9	517	other
2	10/52/30/1	654	requirements/design templates
3	26/43/26/4	675	formal interface specifications
6	22/48/24/6	706	expert person/documentation
1	20/50/22/8	740	formal requirements
4	23/36/23/18	1016	training

It is interesting to note that the faults considered to be prevented by training are the hardest to fix. The formal methods also apply to classes of faults that take a long time to fix.

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

In the Chi-Square analysis, faults prevented by "application walk-throughs," "guideline enforcement," and "other" tended to take fewer days to fix than expected, whereas faults prevented by "formal requirements," "requirements/design templates," and "expert person/documentation" took longer to fix than expected.

In Table 25-15, we present the cross-tabulation of faults and their means of prevention. Again, the faults are represented by the rows, and the means of prevention are represented by the columns. The data is analogous to the preceding cross-tabulation of faults and underlying causes.

For the sake of brevity, we consider only the most frequently occurring faults and their major means of prevention. "Application walk-throughs" were felt to be an effective means of preventing these most significant faults. "Expert person/documentation," "formal requirements," and "formal interface specifications" were also significant means of preventing these faults.

internal functionality (fault 5)

> "application walk-throughs" (prevention 5) were thought to be the most effective means of prevention, applicable to 27% of the occurrences of this fault; "expert person/documentation" (prevention 6) was felt to be the second most effective means, applicable to 18% of the fault occurrences; and "requirements/design templates" were thought to be applicable to 14% of the fault occurrences, a percentage higher than expected.

TABLE 25-15. Cross-tabulating faults and means of prevention

		1	2	3	4	5	6	7	8	9	10
		8.8%	**10.0%**	**7.2%**	**2.2%**	**24.5%**	**15.7%**	**1.5%**	**13.3%**	**9.9%**	**6.9%**
1 language pitfalls	**3.5%**	.0	.1	.1	.0	1.0	.3	.1	1.3	.4	.2
2 protocol	**.4%**	.1	.2	.0	.0	.1	.0	.0	.0	.0	.0
3 low-level logic	**7.9%**	.1	.0	.1	.2	2.3	.3	.2	3.2	.8	.7
4 CMS complexity	**.3%**	.0	.0	.0	.0	.0	.1	.0	.1	.1	.0
5 internal functionality	**25.0%**	1.9	3.5	1.5	.4	6.6	4.4	.2	3.3	3.1	.1
6 external functionality	**3.6%**	.6	.3	.4	.0	.1	.7	.0	.5	.9	.1
7 primitives misused	**2.4%**	.1	.1	.2	.0	.8	.3	.0	.1	.2	.6

		1	2	3	4	5	6	7	8	9	10
		8.8%	**10.0%**	**7.2%**	**2.2%**	**24.5%**	**15.7%**	**1.5%**	**13.3%**	**9.9%**	**6.9%**
8 primitives unsupported	**.6%**	.1	.0	.0	.0	.3	.0	.0	.0	.1	.1
9 change coordination	**4.9%**	.4	.9	.3	.4	.8	.3	.3	.3	.7	.5
10 interface complexity	**11.4%**	2.1	.3	2.1	.0	3.0	1.7	.1	1.2	.7	.2
11 design/ code complexity	**7.7%**	.8	.5	.1	.4	2.2	2.4	.2	.3	.4	.4
12 error handling	**3.3%**	.2	.2	.3	.1	.6	.6	.0	.4	.5	.4
13 race conditions	**4.3%**	.8	.0	.4	.0	1.2	.4	.2	.4	.2	.7
14 performance	**.9%**	.0	.0	.0	.2	.2	.3	.0	.0	.0	.2
15 resource allocation	**1.5%**	.1	.1	.1	.0	.3	.3	.0	.3	.3	.0
16 dynamic data design	**.3%**	.0	.0	.0	.0	.1	.0	.0	.1	.0	.1
17 dynamic data use	**2.1%**	.0	.0	.2	.0	.8	.5	.0	.5	.0	.1
18 static data design	**1.0%**	.1	.1	.0	.0	.2	.2	.0	.0	.3	.1
19 unknown interactions	**.7%**	.1	.0	.2	.0	.0	.2	.0	.0	.2	.0
20 unexpected dependencies	**8.0%**	.6	2.2	1.1	.1	2.3	.6	.0	.4	.6	.1
21 concurrent work	**4.4%**	.4	.7	.0	.2	1.2	1.1	.1	.3	.0	.4
22 other	**5.8%**	.3	.8	.1	.2	.4	1.0	.1	.6	.4	1.9

interface complexity (fault 10)

Again, "application walk-throughs" were considered to be the most effective, applicable to 26% of the cases; "formal requirements" (prevention 1) and "formal interface specifications" were felt to be equally effective, with each preventing 18% of the fault occurrences (in both cases, a percentage higher than expected).

unexpected dependencies (fault 20)

"application walk-throughs" were felt to be the most effective means of preventing this fault, applicable to 29% of the occurrences; "requirements/design templates" were considered the second most effective and applicable to 28% of the fault occurrences (a percentage higher than expected); and "formal interface specifications" were considered applicable to 14% of the fault occurrences, a percentage higher than expected.

design/code complexity (fault 11)

"expert person/documentation" was felt to be the most effective means of preventing this fault, applicable to 31% of the cases (higher than expected); "application walk-throughs" were the second most effective means, applicable to 29% of the occurrences; and "formal requirements" was third, applicable to 10% of the fault occurrences.

Again, for the sake of brevity, we consider only the most frequently occurring means of prevention and the faults to which they were most applicable. Not surprisingly, these means were most applicable to "internal functionality" and "interface complexity," the most prevalent faults. Counterintuitively, they are also strongly recommended as applicable to "low-level logic."

application walk-throughs (prevention 5)

"internal functionality" (fault 5) was considered as the primary target in 27% of the uses of this means of prevention; "interface complexity" (fault 10) was felt to be the secondary target, representing 12% of the uses of this means; and "low-level logic" (fault 3) and "unexpected dependencies" (fault 20) were next with 9% each.

expert person/documentation (prevention 6)

Again, "internal functionality" is the dominant target for this means, representing 29% of the possible applications; "design/code complexity" is the second most applicable target, representing 15% of the possible applications (a percentage higher than expected); and "interface complexity" represented 11% of the uses (higher than expected).

guideline enforcement (prevention 8)

"internal functionality" and "low-level logic" were the dominant targets for this means of prevention, representing 25% and 24%, respectively (the latter being higher than expected); "language pitfalls" (fault 1) was seen as the third most relevant fault, representing 10% of the possible applications (higher than expected); and "interface complexity" was the fourth with 9% of the possible applications of this means of prevention.

Underlying causes and means of prevention

In Table 25-16, it is interesting to note that in the Chi-Square analysis there are lots 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). 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.

TABLE 25-16. Cross-tabulating means of prevention and underlying causes

		1	2	3	4	5	6	7	8	9	10
		20.5%	5.4%	2.0%	25.2%	9.8%	7.3%	17.8%	1.1%	6.8%	4.1%
1 formal requirements	8.8%	.4	2.3	.9	3.5	.8	.3	.5	.1	.0	.0
2 reqs/design templates	10.0%	.4	1.7	.1	3.7	1.9	.1	.8	.0	1.3	.0
3 formal interface specs	7.2%	.8	.3	.1	2.7	.8	.3	2.0	.0	.2	.0
4 training	2.2%	.4	.0	.1	.7	.1	.3	.6	.0	.0	.0
5 application walk-thrus	24.5%	7.5	.2	.3	7.3	3.1	1.8	3.1	.0	.5	.7
6 expert person/doc	15.7%	1.5	.4	.4	3.5	1.8	1.0	5.8	.6	.3	.4
7 design/code currency	1.5%	.4	.0	.0	.6	.2	.1	.2	.0	.0	.0
8 guideline enforcement	13.3%	4.0	.1	.0	.6	.2	1.6	2.5	.0	3.7	.6
9 better test planning	9.90%	2.8	.2	.0	1.7	.8	1.6	1.9	.3	.2	.4
10 others	6.9%	2.3	.2	.1	.9	.1	.2	.4	.1	.6	2.0

We first summarize the means of prevention associated with the major underlying causes. "Application walk-throughs," "expert person/documentation," and "guideline enforcement" were considered important in addressing these major underlying causes.

incomplete/omitted design (cause 4)

"application walk-throughs" (prevention 5) was thought to be applicable to 28% of the faults with this underlying cause (a percentage higher than expected); "requirements/ design templates" (prevention 2) and "expert person/documentation" (prevention 6) were

next in importance with 14% each (the first being higher than expected); and "formal requirements" (prevention 1) was felt to be applicable to 12% of the faults with this underlying cause (a percentage higher than expected).

none given (cause 1)

Again, "application walk-throughs" was thought to be applicable to 37% of the faults with these underlying causes; "guideline enforcement" (prevention 8), "better test planning" (prevention 9), and "other" (prevention 10) were felt to be applicable to 19%, 14%, and 10% of the faults, respectively. In all four of these cases, the percentages were higher than expected.

lack of knowledge (cause 7)

"expert person/documentation" was thought to be applicable to 32% of the faults with this underlying cause, a percentage higher than expected; "application walk-throughs," "guideline enforcement," and "formal interface specifications" were felt to be applicable to 17%, 14%, and 11% of the faults with this underlying cause, respectively, though "application walk-throughs" had a lower percentage than expected, whereas "formal interface specifications" had a higher percentage than expected.

The following summarizes the major underlying causes addressed by the most frequently considered means of prevention. "Lack of knowledge," "none given," "incomplete/omitted design," and "ambiguous design" were the major underlying causes for which these means of prevention were considered important. It is somewhat non-intuitive that the "none given" underlying cause category is so prominent as an appropriate target for these primary means of prevention.

application walk-throughs (prevention 5)

"none given" (cause 1) and "incomplete/omitted design" (cause 4) were thought to be the appropriate for this means of prevention for 31% and 30% of the cases, respectively (higher than expected); "ambiguous design" (cause 5) and "lack of knowledge" (cause 7) both were felt to apply to 13% of the cases (though the first was higher than expected and the second lower).

expert person/documentation (prevention 6)

"lack of knowledge" was considered the major target for this means of prevention, accounting for 37% of the cases (a higher than expected value); "incomplete/omitted design" and "ambiguous design" were thought to be appropriate in 23% and 11% of the cases, respectively; and "none given" was thought appropriate in 10% of the cases (lower than expected).

guideline enforcement (prevention 8)

"none given" and "incorrect modification" were felt to be the most appropriate for this means of prevention for 30% and 28% of the cases, respectively (both higher than expected); "lack of knowledge" and "incorrect earlier fix" were appropriate in 19% and 12% of the cases, respectively (the latter was higher than expected).

Interface Faults Versus Implementation Faults

The definition of an interface fault that we use here is that of Basili and Perricone [Basili and Perricone 1984] and Perry and Evangelist [Perry and Evangelist 1985], [Perry and Evangelist 1987]: 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; remember that some of the "design/code complexity" faults were considered preventable by formal interface specifications. Table 25-17 shows our interface versus implementation fault comparison.

TABLE 25-17. 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.

Table 25-18 compares 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.

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

TABLE 25-18. Interface/implementation faults and underlying causes

		Interface	Implementation
		49%	**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%

		Interface	Implementation
		49%	**51%**
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 25-19 compares interface and implementation faults with respect to the means of prevention. Not surprisingly, means 1 and 3 were more applicable to interface faults than to implementation faults. Means of prevention 8, 4, and 6 were considered more applicable to implementation faults than to interface faults.

TABLE 25-19. Interface/implementation faults and means of prevention

		Interface	Implementation
		49%	**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%

What Should You Believe About These Results?

Designing empirical studies is just like designing software systems: it is impossible to create a bug-free system, and we often make design mistakes and create systems with flaws and weaknesses. The main question in both cases is: do the problems negate the usefulness of the software systems or the empirical studies?

There are three main questions we need to address in order to determine how good our study is and whether you can justifiably use our results: 1) are we measuring the right things, 2) are there other things that might be the explanations for what we see (i.e., did we do it right?), and 3) what do our results apply to (i.e., what can we do with the results)?

Are We Measuring the Right Things?

We believe that we have a very strong argument to support our claim that we have addressed the critical issues in understanding software development faults and their implications. We address the fundamental issues in fault studies: the faults that occur, how hard it is to find and fix them, their underlying causes, and how might we prevent them, detect them, or ameliorate them. In addition, we addressed a question raised in response to the interface fault studies we had done earlier: which are harder to find and/or fix, interface or implementation faults? Strong support for this comes from the consistency and mutual support provided by the Chi-Square analysis, in which there are very strong relationships between the faults detected, their underlying causes, and their means of prevention. These strong relationships indicate a consistent understanding of the various parts of the survey and their interrelationships.

There are, however, several weaknesses that need to be addressed. First, the fault categories are poorly constructed. Second, the find and fix scales are not identical, with the fix scale being much better than the find scale. Third, the line between interface and implementation faults is not cleanly drawn.

The primary strength of the fault type list is that it was drawn up by the developers themselves, not the researchers. The weakness is that the list is basically unstructured and too long. There may be a tendency to pick the first thing that comes close rather than search the list exhaustively to find the best match.

In a subsequent study [Leszak et al. 2000], [Leszak et al. 2002], we corrected the fault type problem by partitioning the fault list into three categories: implementation, interface, and external faults (which also solved the third weakness mentioned above). Under each of these three fault categories were then between six and eight fault subcategories appropriate for each fault category (see page 177 of [Leszak et al. 2002]).

The scales used for finding and fixing faults was again the choice of the software developers, but had more serious consequences than poor structuring. The scale used for fixing faults is intuitively one that can be used easily, as it is easy to remember if something took less than a day, week, or month, which makes this measure much less likely to be misclassified. The scale for finding a fault, on the other hand, was qualitative rather than quantitative and much more likely to be subjective with individual variance (which we were not able to determine, because of management restrictions). Our recommendation is to use the quantitative scale for effort in terms of time for both scales. Indeed we did do that as well in subsequent studies [Leszak et al. 2000], [Leszak et al. 2002].

The separation of faults into interface and implementation faults was not a completely clean one, as some of the fault categories counted as interface may have included some implementation faults as well (and vice versa). So our distinction between the two is approximate at best. As mentioned earlier, we later solved that problem by structuring the separation of faults into implementation, interface, and external [Leszak et al. 2000], [Leszak et al. 2002].

Did We Do It Right?

In both phases, approximately 68% of questionnaires were returned—that is, we have data on about two-thirds of the MRs in both the overall survey and in the design/coding survey. Given the circumstances under which the survey was taken, this level of response exceeded our best expectations. Indeed, this factor of a large number of responses alone provides an argument for having data that can be relied upon.

As we cannot give the hard numbers as part of our report, we have tried to indicate the level of responses via the precision we used in discussing the results. Given the amount of data, we could have easily justified using two decimal places in reporting the data instead of the one decimal place we used for ease of understanding the data.

As with all surveys, there is the unanswerable question of how those who did not respond would have affected the results. Fortunately, we know of no existent factors (such as reporting only the hard or easy problems, receiving reports from only junior or senior programmers, etc.) that would have skewed the results in any way [Basili and Hutchens 1983].

We mentioned earlier that there were significant constraints placed on the study by project management: first, the study had to be completely nonintrusive; second, it had to be strictly voluntary; and third, it had to be completely anonymous. Because of these management mandates, we were unable to validate the results [Basili and Weiss 1984] and are unable to assess the accuracy of the responses. Mitigating the lack of validation are two facts: first, the questionnaire was created by the authors working with a group of developers; second, the questionnaire was reviewed by an independent group of developers. Since the purpose of the post-survey validation is understanding the level at which those surveyed understood the survey properly, we believe that our pre-survey efforts provide a useful and valid alternative because 1) we ensured that the survey was the language used by the developers themselves by their participation in its development, and 2) we pretested the survey successfully with a small group of developers, and no misunderstandings arose in the pretests.

The remaining problem is raised by the fact that there was a lapse time of up to a year between closing the MR and filling out the survey. Thus, there is a possibility of information loss due to the time lapse between solving the problem and describing it. However, having the person who was in charge of the problem at time of closure is still much better than having someone who had no involvement in the problem interpreting the MR for the survey. This lack of

"freshness" could of course be resolved by making the fault survey part of the normal process of closing an MR.

One remaining caveat: the overall proportions of the faults may be affected by the fact that data is kept only during the testing phase of evolution. MRs for the entire process from the receipt of the requirements to the release of the system would, of course, give a much more accurate picture. We note, however, that this approach of keeping track of faults only once testing has begun is pretty much standard for most software developments and therefore only a very minor issue.

On the whole, we believe that the few problems with our empirical design are significantly outweighed by the evidence supporting our claim that our data is valid and that there are no other factors responsible for our results.

What Can You Do with the Results?

The main question for any empirical study is: "what do these results mean for me in the context of my work as a software developer?" Part of that answer depends on how representative the study is, and there are two different ways of answering that question.

The first way is to ask the question: "how representative is this release in the context of all the releases for this system that has been studied?" If it is not representative of the system and its various releases, then its general usefulness is not clear. In this case, we claim that it is representative because the mix of fault fixes, new features, and improvements was the same as for previous releases. For the first few releases after this one, however, there was an increased emphasis on removing faults before the previous mix of corrective, adaptive, and perfective changes was resumed.

Given a positive answer to the first question, then the second way to answer this question is to ask: "how representative is this release of this system of other software systems?" With respect to other large-scale, highly fault-tolerant, ultra-reliable real-time systems, this release would represent this small class of systems in that it is built and evolved in the context of a commonly used Unix Development Environment using a commonly used programming language such as C. One would expect to see similar kinds of problems in such systems.

How relevant is it to the development of other types of software systems? We would claim that it is highly relevant. Look at the top five fault types. There is nothing there that would lead one to believe that the main problems were domain specific. Indeed the entire list of faults, with a few exceptions, would be the kinds of things found in pretty much any software system development, whatever the domain or size of the project. We would further claim that there is nothing in the list of underlying causes that would preclude the vast variety of other types of software developments. We would make a similar claim for the means of prevention. The primary differences would be in the observed frequencies of the various faults, causes, and means of preventions.

The design of the study is certainly applicable, no matter what the size or domain of the software system being developed. The data itself is also applicable if you find you have the same frequently observed problems. There is an internal consistency to the data and their interrelationships that supports this claim.

What Have We Learned?

The results of the two studies are summarized as follows:

- Problems with requirements, design, and coding accounted for 34% of the total MRs. Requirements account for about 5% of the total MRs and, although not extremely numerous, are particularly important because they have been found so late in the development process, a period during which they are particularly expensive to fix.
- Testing large, complex real-time systems often requires elaborate test laboratories that are themselves large, complex real-time systems. In the development of this release, testing-related MRs accounted for 25% of the total MRs.
- The fact that 16% of the total MRs are "no problems" and the presence of a significant set of design and coding faults such as "unexpected dependencies" and "interface and design/code complexity" indicate that *lack of system knowledge* is a significant problem in the development of this release.
- Of the design and coding faults, 78% took five days or less to fix; 22% took six or more days to fix. We note that there is a certain overhead factor that is imposed on the fixing of each fault that includes getting consensus, building the relevant pieces of the system, and using the system test laboratory to validate the repairs. Unfortunately, we do not have data on those overhead factors.
- Five fault categories account for 60% of the design and coding faults: internal functionality, interface complexity, unexpected dependencies, low-level logic, and design/code complexity. With the exception of "low-level logic," this set of faults is what we expect would be significant in evolving a large, complex real-time system.
- Weighting the fault categories by the effort to find and to fix them yielded results that coincide with our intuition of which faults are easy and hard to find and fix.
- "Incomplete/omitted design," "lack of knowledge," and "none given" (which we interpret to mean that sometimes we just make a mistake with no deeper, hidden underlying cause) account for the underlying causes for 64% of design and coding faults. The weighting of the effort to fix these underlying causes coincides very nicely with our intuition: faults caused by requirements problems require the most effort to fix, whereas faults caused by ambiguous design and lack of knowledge were among those that required the least effort to fix.
- "Application walk-throughs," "expert person/documentation," "guideline enforcement," and "requirements/design templates" represent 64% of the suggested means of preventing

design and coding faults. As application walk-throughs accounted for 25% of the suggested means of prevention, we believe that this supports Curtis, Krasner, and Iscoe's claim [Curtis et al. 1988] that lack of application knowledge is a significant problem.

- Although informal means of prevention were preferred over formal means, it was the case that informal means of prevention tended to be suggested for faults that required less effort to fix and formal means tended to be suggested for faults that required more effort to fix.
- In Perry and Evangelist [Perry and Evangelist 1985], [Perry and Evangelist 1987], interface faults were seen to be a significant portion of the entire set of faults (68%). However, there was no weighting of these faults versus implementation faults. We found in this study that interface faults were roughly 49% of the entire set of design and coding faults and that they were harder to fix than the implementation faults (see the previous discussion). Not surprisingly, formal requirements and formal interface specifications were suggested as significant means of preventing interface 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 the data point 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 process.
- Incorporate the fault survey as an integral part of MR closure and gather the fault-related information while it is fresh in the developer's mind. This data provides the basis for measurement-based process improvement where the current most frequent or most costly faults are remedied.
- Incorporate the informal, people-intensive means of prevention into the current process (such as application walk-throughs, expert person or documentation, guideline enforcement, etc.). As our survey has shown, this will yield benefits for the majority of the faults reported here.
- Introduce techniques and tools to increase the precision and completeness of requirements, architecture, and design documents. 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.

We close with several lessons learned that may go a long way toward the improvement of future system developments:

- The fastest way to product improvement as measured by reduced faults is to hire people who are knowledgeable about the domain of the product. Remember, lack of knowledge tended to dominate the underlying causes. The fastest way to increase the knowledge needed to reduce faults is to hire knowledgeable people.

- One of the least important ways to improve software developments is to use a "better" programming language. We found relatively few problems that would have been solved by the use of better programming languages.
- Techniques and tools that help to understand the system and the implications of change should be emphasized in improving a development environment. Remember that knowledge-intensive activities tended to dominate the means of prevention.

Acknowledgments

My special thanks to Carol Steig for her earlier work with me on this project. David Rosik contributed significantly to the general MR survey; Steve Bruun produced the cross-tabulated statistical analysis for the design/coding survey and contributed, along with Carolyn Larson, Julie Federico, H. C. Wei, and Tony Lenard, to the analysis of the design/coding survey; and Clive Loader increased our understanding of the Chi-Square analysis. We especially thank Marjory P. Yuhas and Lew G. Anderson for their unflagging support of this work. And finally, we thank all those who participated in the survey.

References

[Basili and Hutchens 1983] Basili, Victor R. and David H. Hutchens. 1983. An Empirical Study of a Syntactic Complexity Family. *IEEE Transactions on Software Engineering* SE-9(6):664–672.

[Basili and Perricone 1984] Basili, Victor R. and Barry T. Perricone. 1984. Software Errors and Complexity: an Empirical Investigation. *Communications of the ACM* 27(1):42–52.

[Basili and Weiss 1984] Basili, Victor R. and David M. Weiss. 1984. A Methodology for Collecting Valid Software Engineering Data. *IEEE Transactions on Software Engineering* SE-10(6): 728–738.

[Boehm 1981] Boehm, Barry W. 1981. *Software Engineering Economics*. Englewood Cliffs, NJ: Prentice-Hall.

[Bowen 1980] Bowen, John B. 1980. Standard Error Classification to Support Software Reliability Assessment. *Proceedings of the AFIPS Joint Computer Conferences, 1980 National Computer Conference*:697–705.

[Brooks 1995] Brooks, Jr., Frederick P. 1995. *The Mythical Man-Month: Essays on Software Engineering*, Anniversary Edition. Addison-Wesley.

[Curtis et al. 1988] Curtis, Bill, Herb Krasner, and Neil Iscoe. 1988. A Field Study of the Software Design Process for Large Systems. *Communications of the ACM* 31(11):1268–1287.

[Endres 1975] Endres, Albert. 1975. An Analysis of Errors and Their Causes in System Programs. *IEEE Transactions on Software Engineering* SE-1(2):140–149.

[Fenton Ohlsson 2000] Fenton, N.E, and N. Ohlsson. 2000. Quantitative Analysis of Faults and Failures in a Complex Software System. *IEEE Transactions on Software Engineering* SE-26(8):797–814.

[Glass 1981] Glass, Robert L. 1981. Persistent Software Errors. *IEEE Transactions on Software Engineering* SE-7(2):162–168.

[Graves et al. 2000] Graves, T.L., A.F. Karr, J.S. Marron, and H. Siy. 2000. Predicting Fault Incidence Using Software Change History. *IEEE Transactions on Software Engineering* SE-26(7): 653–661.

[Lehman and Belady 1985] Lehman, M. M. and L. A. Belady. 1985. *Program Evolution: Processes of Software Change*. London: Academic Press.

[Leszak et al. 2000] Leszak, Marek, Dewayne E. Perry, and Dieter Stoll. 2000. A Case Study in Root Cause Defect Analysis. *Proceedings of the 22nd International Conference on Software Engineering*:428–437.

[Leszak et al. 2002]Leszak, Marek, Dewayne E. Perry, and Dieter Stoll. 2002. Classification and Evaluation of Defects in a Project Retrospective. *Journal of Systems and Software* 61(3):173–187.

[Musa et al. 1987]Musa, J. D., A. Jannino, and K. Okumoto. 1987. *Software Reliability*. New York: McGraw-Hill.

[Ostrand and Weyuker 1984] Ostrand, Thomas J. and Elaine J. Weyuker. 1984. Collecting and Categorizing Software Error Data in an Industrial Environment. *The Journal of Systems and Software*, 4(4):289–300.

[Ostrand and Weyuker] Ostrand, Thomas J. and Elaine J. Weyuker. 2002. The Distribution of Faults in a Large Industrial Software System. *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*:55–64.

[Ostrand et al. 2005] Ostrand, T.J., E.J. Weyuker, and R.M. Bell. 2005. Predicting the Location and Number of Faults in Large Software Systems. *IEEE Transactions on Software Engineering* SE-31(4):340–355.

[Perry and Evangelist 1985] Perry, Dewayne E. and W. Michael Evangelist. 1985. An Empirical Study of Software Interface Errors. *Proceedings of the International Symposium on New Directions in Computing*:32–38.

[Perry and Evangelist 1987] Perry, Dewayne E. and W. Michael Evangelist. 1987. An Empirical Study of Software Interface Faults—An Update. *Proceedings of the Twentieth Annual Hawaii International Conference on Systems Sciences* II:113–126.

[Perry et al. 2001] Perry, Dewayne E., Harvey P. Siy, and Lawrence G. Votta. 2001. Parallel Changes in Large Scale Software Development: An Observational Case Study. *Transactions on Software Engineering and Methodology* 10(3):308–337.

[Perry and Steig 1993] Perry, Dewayne E. and Carol S. Steig. 1993. Software Faults in Evolving a Large, Real-Time System: a Case Study. In *Lecture Notes in Computer Science, Volume 717: Proceedings of the 4th European Software Engineering Conference*, ed. I. Sommerville and M. Paul, 48–67.

[Perry and Wolf 1992] Perry, Dewayne E. and Alexander L. Wolf. 1992. Foundations for the Study of Software Architecture. *ACM SIGSOFT Software Engineering Notes* 17(4):40–52.

[Purushothaman and Perry 2005] Purushothaman, Ranjith and Dewayne E Perry. 2005. Toward Understanding the Rhetoric of Small Source Code Changes. *IEEE Transactions on Software Engineering* SE-31(6):511–526.

[Rowland et al. 1983] Rowland, B. R., R. E. Anderson, and P. S. McCabe. 1983. The 3B20D Processor & DMERT Operating System: Software Development System. *The Bell System Technical Journal* 62(1/2):275–290.

[Shao et al. 2007] Shao, D., S. Khurshid, and D. Perry. 2007. Evaluation of Semantic Interference Detection in Parallel Changes: an Exploratory Experiment. *Proceedings of the 23rd IEEE International Conference on Software Maintenance*:74–83.

[Siegel et al. 1988] Siegel, Sidney and N. John Castellan, Jr. 1988. *Nonparametric Statistics for the Behavioral Sciences*, Second Edition. New York: McGraw-Hill.

[Schneidewind and Hoffmann 1979] Schneidewind, N. F. and Heinz-Michael Hoffmann. 1979. An Experiment in Software Error Data Collection and Analysis. *IEEE Transactions on Software Engineering* SE-5(3):276–286.

[Thayer et al. 1978] Thayer, Thomas A., Myron Lipow, and Eldred C. Nelson. 1978. Software Reliability—A Study of Large Project Reality. In *TRW Series of Software Technology*, Volume 2. Amsterdam: North-Holland.

[Thione and Perry 2005] Lorenzo Thione, G. and Dewayne E. Perry. 2005. Parallel Changes: Detecting Semantic Interferences. *Proceedings of the 29th Annual International Computer Software and Applications Conference*:47–56.