An Empirical Study of Software Interface Faults

D.E. Perry

AT&T Bell Laboratories
Murray Hill, NJ 07974

W.M. Evangelist

AT&T Bell Laboratories
Naperville, IL 60566

Abstract

We demonstrate through a survey of the literature on software errors that the research community has paid little attention to the problem of interface errors. The main focus of the paper is to present the results of a preliminary empirical study of error reports for a large software system. We determined that at least 66% of these errors arose from interface problems. The errors fell naturally into fifteen separate categories, most of which were related to problems with the methodology.

1. Introduction and Background.

This paper represents a "work in progress" report on interface errors, and the primarily methodological problems from which they stem, in large, real-time systems. It has been our intuition, derived from experience with the construction of numerous large systems, that the points of contact among the various components (that is, the *interfaces*) in large systems represent a significant set of problems both in the development and in the evolution of these systems. Thus, a primary purpose of this report is to describe preliminary, mostly qualitative, results from an empirical study of a large real-time system that support this intuition. In addition to arguing the significance of the problem, we outline future research designed to help provide further insight into its nature.

We first present, in section 1.1, an overview of previous work on measuring, evaluating, and categorizing errors in the development of large systems. In section 1.2, we characterize the system from which our errors are drawn and define the criteria for choosing our set of errors and error reports. In section 2, we present our data and analyze it. Finally, in section 3, we summarize the results and point to future work.

1.1 Introduction.

There exist in the open literature relatively few studies of the errors encountered in the development of large systems. One of the first papers to analyze errors in a large program (the operating system DOS/VS) was that of Endres[1]. This paper reports on errors detected during the internal testing of approximately 500 modules having an average size of 360 executable lines of code. Endres based his error classification on the two primary activities used to produce the system: designing and implementing the algorithms. Although interface problems were not a specifically cited category, aspects of interface problems were identified within various different subcategories.

Thayer, Lipow, and Nelson[2] provide an extensive categorization of errors and a discussion of several large projects. While their definition of an interface error is substantially narrower than ours (see below), they do include interface problems as one of their nine major categories. In two of the projects reported on, the interface error categories provide a significant number of errors (page 59: 17% and 22.5% for projects 3 and 4 respectively). Bowen[3] reports on projects from Hughes-Fullerton using an error scheme based on the TRW categorization, but with a much smaller percentage of interface errors (4.5%).

Schneidewind and Hoffmann[4] categorize errors according to their occurrence in a life-cycle phase and do not explicitly distinguish a class of interface errors, although several "coding errors" can be identified as typical interface problems. Glass[5] reports on errors that are not discovered until late in the development of several large systems. The errors are grouped into twelve different categories, the largest of which was "omitted logic", and one catch-all category; interface errors are omitted from the discussion. Ostrand and Weyuker[6] report on the development of an interactive special-purpose editor and introduce an attribute categorization scheme (category, type, presence, and use). Although they do discuss inadequate specifications, it is unclear whether they are concerned about requirements specifications or interface specifications. Interface errors, as such, are not considered.

Finally, Basili and Perricone[7] provide an analysis of change data collected during the development of a medium-scale system (approximately 90,000 lines of code). We accept their definition of *interface errors* as "those that are associated with structures existing outside the module's local environment but which the module used" ([7], page 47), but we prefer the term *interface fault* to describe this concept. (*Interface errors* are the human mistakes that lead to the faults.) Since 39% of the errors studied were interface errors in their sense of the term, Basili and Perricone conclude that "interfaces appear to be a major problem" ([7], page 48 and 47).

This view of interface problems as the root cause of a significant number of faults in large systems underlies the following presentation of criteria, data, and analysis.

1.2 Background for the study.

The software studied formed the third release of a real-time system. The life-cycle used to develop this software included a specification of requirements, high-level design, detailed design, code implementation, unit test, integration test, and system test, in that order. We examined a selection of faults reported by testers on approximately 350,000 non-commentary source lines from files written in the **C** programming language. (The number of non-commentary source lines in a file is the number of newlines remaining after comments and blank lines are deleted.) We also included fault reports written against a selection of global header files. (Of course, all faults reported during testing were removed before system release.)

Faults discovered during testing are reported and tracked by a modification request tracking system. Access to submitted source files is possible only through the tracking system. Thus, all fault repair activity is tracked. (One problem, however, is that occasionally more than one fault is repaired in a file that has been accessed by specifying a single fault report.) Modification requests (MRs) include both software faults, our primary interest here, and feature requests (enhancements for the new release of the system). In the sequel, the term MR will refer exclusively to software faults for which either an implementation file (one with a ".c" suffix) or a global header file (one with a ".h" suffix) had to be modified.

For the purpose of partitioning the set of MRs into interface and non-interface faults, we used the following operational definition: an MR is an *interface MR* if it

i. requires a change in two or more **C** files, or

ii. requires a change in a global header file.

MRs falling under criterion i. are interface MRs in the sense that information had to be altered in more than one independent software unit. MRs falling under criterion ii. are interface problems for the reason that global header files in **C** are the principal mechanism for specifying interfaces among independent software entities.

If this operational definition captures the meaning of the term "interface fault," we can demonstrate the importance of studying the problem. Approximately 30% of the MRs in the database satisfied one or both of the criteria given. A close examination of a significant sample of these MRs confirmed the adequacy of the operational definition—virtually all of the MRs were arguably interface problems. The reader should understand that files are the smallest entity tracked by the MR tracking system. Interface faults, however,

could occur among separate functions within a file, and some interface problems could have emerged and been repaired before the software was tested as an integral unit. Furthermore, many interface faults affecting two or more files require modification of only one of the files. A preliminary analysis of those MRs that had an impact on one file only confirmed this observation: more than 52% reflected interface problems among multiple files. (The combined percentage for both the latter group and the group satisfying the operational definition is 66%.) We must, therefore, regard the 30% figure as a very coarse lower bound on the percentage of software faults caused by interface errors. It is not hard to conclude that the interface problem is significant and justifies increased attention.

To obtain a sample of tractable size for detailed examination, we used a random process to select 94 MRs. Of these, 85 MR descriptions provided sufficient information to support an analysis of their characteristics. Each of these 85 MR descriptions contained a statement of the fault, as perceived by the tester who initiated the MR, as well as a statement of the repair activity undertaken by the developer responsible for the affected files. A primary purpose for analyzing this sample is to describe some ways in which the interface problem contributes materially to the rapidly escalating costs of software development. Another purpose is to derive preliminary suggestions for environmental changes to minimize the occurrence of interface errors. Finally, the results of the analysis demonstrate that the problem should be studied in greater depth.

2. Data Presentation and Analysis.

2.1 The data.

We classified the interface faults represented by the randomly selected data by studying each MR description in detail to determine the underlying interface problem. As each MR was analyzed, it was placed in either a previously constructed category or in a new category, the existence of which had to be justified. Our goal was to construct the least number of categories consistent with a transparent presentation of the results of the analysis.

The categories dictated by the data were as follows:

1. *Construction.* These are interface faults endemic to languages that physically separate the interface specification from the implementation code. For example, in the **C**-based system under study, problems associated with using the **#include** directive caused many of the MRs in this category.

2. *Inadequate functionality.* These are faults caused by the fact that some part of the system assumed, perhaps implicitly, a certain level of functionality not provided by another part of the system.

3. *Disagreements on functionality.* These are faults caused, most likely, by a dispute over the proper location of some functional capability in the software.

4. *Changes in functionality.* A change in the functional capability of some unit was made in response to a changing need.

5. *Added functionality.* A completely new functional capability was recognized and requested as a system modification.

6. *Misuse of interface.* These are faults arising from a misunderstanding of the required interface among separate units.

7. *Data structure alteration.* Either the size of a data structure was inadequate or it failed to contain a sufficient number of information fields.

8. *Inadequate error processing.* Errors were either not detected or not handled properly.

9. *Additions to error processing.* Changes in other units dictated changes in the handling of errors.

10. *Inadequate postprocessing.* These faults reflected a general failure to free the computational workspace of information no longer required.

11. *Inadequate interface support.* The actual functionality supplied was inadequate to support the specified capabilities of the interface.

12. *Initialization/value errors.* A failure to initialize or assign an appropriate value to a data structure caused these faults.

13. *Violation of data constraints.* A specified relationship among data items was not supported by the implementation.

14. *Timing/performance problems.* These faults were caused by inadequate synchronization among communicating processes.

15. *Coordination of changes.* Someone failed to communicate modifications to one software unit to those responsible for other units that depend on the first.

A bar graph presents the breakdown of the 85 MRs into the categories listed. (This graph and all tables appear at the end of the paper.)

2.2 Analysis.

The most significant categories in the data are: inadequate error processing (15.1%), construction and inadequate functionality (both 12.9%), inadequate postprocessing (10.5%), and data structure alteration (9.4%). Changes in functionality (1.2%) and added functionality and disagreements in functionality (both 2.3%) were the smallest categories.

In the following analysis, we explore likely causes for each of the 15 general categories of faults and offer potential solutions for the errors that caused them. We have emphasized the preliminary nature of this work. We also wish to stress that the causes and solutions given below are intended more as a framework for further exploration of the interface problem than as a definitive program of solutions to it.

1. **Construction.**

   Problem Cause. A problem with constructing header files in **C**, for example, is that it is not always clear what should be included for the file to function independently.

   Potential Solution. Certainly, methodological enforcement of header file independence would attenuate the problem. No general solution can be offered, however, since varying **C** environments

would have varying technical requirements.

2. **Inadequate functionality.**

   Problem Cause. One possible cause is the informality and incompleteness of many design specifications. Another possibility is the developer's inexperience with the current environment.

   Potential Solution. It appears, in the first case, that either the design methodology is inadequate or that it is inadequately enforced. The second case might be addressed through a higher level of training of inexperienced developers.

3. **Disagreements on functionality.**

   Problem Cause. The problem is one of methodology, since these disputes should not occur at the code level. It is also possible that inexperienced personnel contribute to the problem.

   Potential Solution. Preferably, the methodology should be altered to detect such potential disputes at the design stage. When the disputes emerge during coding, the methodology should require feedback to the designers, who must resolve the dispute because of their more global conception of the system.

4. **Changes in functionality.**

   Problem Cause. There are many potential causes, most stemming from incomplete statements of functionality in such documents as needs, requirements, design, and the like.

   Potential Solution. Clearly, a better understanding of the customer's needs, followed by a high traceability of requirements from one level of document to the next, would lessen the opportunity for faults of this kind.

5. **Added functionality.**

   Problem Cause. These faults derive from the long-standing problem of the continuing evolution of system requirements during the construction of the system.

   Potential Solution. The issue of continuing evaluation of system requirements is partly a management and partly a technical problem. In any case, we will not attempt to resolve it here.

6. **Misuse of interface.**

   Problem Cause. Interface specifications were probably not given with sufficient clarity.

   Potential Solution. Perhaps a slight change of emphasis in the methodology, coupled with a greater dependence on formalism, would diminish the impact of these faults.

7. **Data structure alteration.**

   Problem Cause. These are similar in nature to the functionality problems above, but they most likely occurred at the detailed design level.

   Potential Solution. Even though the mistake probably occurred during low-level design, the problem has its roots in the failure of the high-level design to specify fully the required capability of the data

structure. The large number of errors of this type argues for paying careful attention at high-level design to the *capacities* of high-use data structures.

8. **Inadequate error processing.**

   Problem Cause. Incomplete designs could contribute to the lack of error detection. Incomplete understanding of the potential error could lead to handling it improperly.

   Potential Solution. The design method should be modified to detect incompleteness—potential error conditions should be particularly emphasized. Clearly, greater traceability of error side-effects on the rest of the system is desirable.

9. **Additions to error processing.**

   Problem Cause. Either necessary functionality is missing from current error processing that would help trace errors or current techniques of error processing require modification.

   Potential Solution. The issue of appropriate functionality for error processing is difficult to determine in the general case. One partial solution is for the design methodology to emphasize more heavily the design of exception handling procedures.

10. **Inadequate postprocessing.**

    Problem Cause. Low-level design does not fully address the problem.

    Potential Solution. A small change in the methodology to emphasize releasing unneeded computational workspace after a process has ended, particularly message buffers in the system under study, would address this obviously significant problem (10.6% of the total number of MRs).

11. **Inadequate interface support.**

    Problem Cause. Design reviews are inadequate.

    Potential Solution. Use a formal design inspection methodology *and enforce it.* In particular, inspectors should closely examine interfaces among units to determine compatibility. Problem 11 frequently reflected the classical interface problem of one unit erroneously expecting another to conform to some standard—a bug that should be observed at the design stage.

12. **Initialization/value errors.**

    Problem Cause. Problems of this kind are usually caused by simple oversight.

    Potential Solution. Effective code inspections, as well as compilers that enforce initialization, for example, could solve this problem.

13. **Violation of data constraints.**

    Problem Cause. Detailed design specifications are incomplete.

    Potential Solution. A low-level design system that allows formal specification of the constraints, and provides verification that they hold, would solve the problem.

14. **Timing/performance problems.**

    Problem Cause. These problems are often caused by incomplete or inconsistent low-level design specifications. They possibly arise from undue emphasis on controlling the process, rather than the object being manipulated.

    Potential Solution. A more formal low-level design language and system that checks for completeness and consistency should be employed. More balanced emphasis in design philosophy on objects, as well as processes, is desirable.

15. **Coordination of changes.**

    Problem Cause. This is a classical management problem.

    Potential Solution. Apply classical techniques to solve the communications problems.

We mentioned previously that an attempt was made to allow the data to dictate the specific categories of analysis. This method appeared to be well suited to an explication of the data. It also revealed the fact that none of the distinguished categories was dominant. Thus, the evidence would not support corrective efforts that concentrated on a small, arbitrary selection of these categories, since the return on investment would also be small.

(The *Inscape* research project — see Perry[8] for a brief overview — addresses a large number of these problems by providing a program construction and evolution environment based on the constructive use of interface specifications. Module interfaces are described in *Instress* (Inscape's interface specification language) so that the Inscape environment can enforce their consistent use in constructing new components. Specifically, details about data initialization, data constraints, operation pre- and post-conditions, exceptions and their implications, and postprocessing requirements are included as part of the interface. Furthermore, Inscape manages the details of interconnections among constructed components and thus is able to manage the problems of making changes and to guarantee the completeness and consistency of those changes.)

At this point, it may be desirable to use a less powerful microscope and combine conceptually related categories. (See Table 1.) We obtained the new categories by partitioning the set of 15 categories derived from the data into 8 conceptually related subsets. At the higher conceptual level, these more general problem types appear to be significant in approximately the order given in the table. Without regard to the amount of effort required to address the individual categories, the questions of incorrect handling of data and errors and the problems with functionality offer the greatest promise for concentrated work.

One might also wish to organize an attack on the various interface problems by emphasizing those activities most likely to have an impact on these problems. Thus, another useful view of the data may be found in Table 2, which groups the original categories according to similarity of recommended corrective techniques. This grouping is not a partition, of course, since more than one technique might be applied to a particular category. The table very clearly demonstrates that difficulties with methodology underlie the great majority of interface problems. We must conclude that resources spent on a carefully tracked, commonly understood, and generally enforced methodology that addresses the issues raised in our data analysis would have a high payoff.

3. Conclusions and Future Work.

In the introduction to this paper, we showed that the importance of the problem of interface errors has not received adequate recognition. The published literature contains only scanty reference to the question; it is seldom considered in depth by research on the general problem of software errors. Through studying fault reports from a large, real-time system, we have produced evidence that the problem is significant. As many as 66% of the faults studied can be attributed to interface errors.

We analyzed a sufficiently large sample of the fault reports to gain insight into the faults and their probable causes. The analysis identified several general conceptual areas of deficiency. Greater care with data handling, with functionality definition and tracking, and with error handling would help to minimize interface errors. These problems were caused, in large part, by inadequate or unenforced methodology.

Our purpose in presenting these initial, qualitative results is to stimulate research and discussion on what we feel is an important, but understudied, problem. Clearly, numerous questions were either unanswered or studied to an insufficient depth by this preliminary analysis. For example:

1. We currently have no data on the effort needed to repair the faults discussed. Thus, we implicitly based our recommendations on a unit weighting scheme. If we can weight faults by repair effort, we can give a more accurate picture of the best strategy for minimizing interface errors.

2. We used an operational definition of "interface MR" that appeared to produce at least a subset of the set of interface faults. We plan to pursue the study of MRs that did not meet our operational criteria. We wish to determine more precisely the classes of interface MRs that can be fixed through modifying a single **C** file.

3. Without question, it will be necessary to deepen our analysis of "problem causes" by interviewing developers responsible for the system under study, since the on-line database from which we extracted our data contained entries of varying degrees of completeness.

4. Finally, we plan to conduct similar studies of several more large systems of different composition to confirm our results and help identify invariant categories. This latter study is already underway.

Acknowledgements. We wish to thank C.E. Kron, N.E. Gardei, and S.E. Schwab for their help in gathering the data used in this study. The assistance of G.G. Lecompte in data gathering and presentation and in understanding the often cryptic MR descriptions was invaluable. Finally, an anonymous referee made an insightful observation that affected the presentation of material.

REFERENCES

1. Albert Endres, An Analysis of Errors and Their Causes in System Programs, *IEEE Transactions on Software Engineering*, Vol. SE-1, No. 2, (June 1975), 140-149.

2. Thomas A. Thayer, Myron Lipow, and Eldred C. Nelson, *Software Reliability - A Study of Large Project Reality*, TRW Series of Software Technology, Volume 2. North-Holland, 1978.

3. John B. Bowen, Standard Error Classification to Support Software Reliability Assessment, *AFIPS Conference Proceedings, 1980 National Computer Conference*, 1980, 697-705.

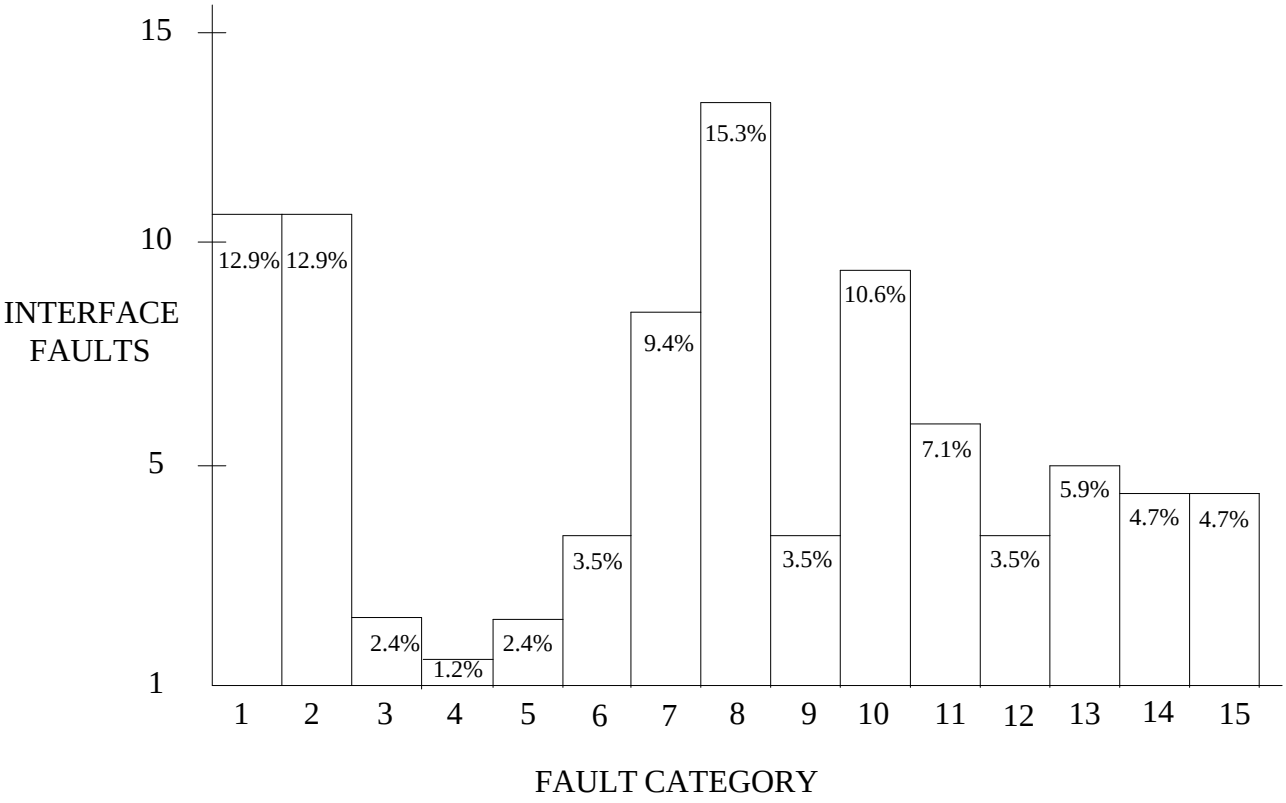
4. N. F. Schneidewind and Heinz-Michael Hoffmann, An Experiment in Software Error Data Collection and Analysis, *IEEE Transactions on Software Engineering*, Vol. SE-5, No. 3, (May 1979), 276-286.

5. Robert L. Glass, Persistent Software Errors, *IEEE Transactions on Software Engineering*, Vol. SE-7, No. 2, (March 1981), 162-168.

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

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

8. D. E. Perry, A Position Paper: The Constructive Use of Interface Specifications, *Third International Workshop on Software Specifications and Design*, IEEE Computer Society, 26-27 August 1985, London, England.



LEGEND: FAULT CATEGORY

1. Construction.
2. Inadequate functionality.
3. Disagreements on functionality.
4. Changes in functionality.
5. Added functionality.
6. Misuse of interface.
7. Data structure alteration.
8. Inadequate error processing.
9. Additions to error processing.
10. Inadequate postprocessing.
11. Inadequate interface support.
12. Initialization/value errors.
13. Violation of data constraints.
14. Timing/performance problems.
15. Coordination of changes.

Figure 1. Faults *vs.* Categories.

Table 1. Conceptual Categories		
Category	*Ref. Nos.**	*Tot. %*
1. data	**7, 12, 13**	**18.8**
2. functionality	**2, 3, 4, 5**	**18.7**
3. error handling	**8,9**	**18.6**
4. construction	**1**	**12.9**
5. inadequate postprocessing	**10**	**10.5**
6. inadequate interface support	**11**	**7.0**
7. timing	**14**	**4.7**
8. coordination	**15**	**4.7**

*** See section 2.1.**

Table 2. Categories of Correction		
Category	*Ref. Nos.**	*Tot. %*
1. general methodology problems	**2, 3, 6, 7, 8, 9, 10, 12, 13, 14**	**73.1**
2. methodology tools	**8, 10, 12, 13, 14**	**39.5**
3. methodology enforcement	**1, 2, 11**	**35.8**
4. traceability enforcement	**4, 8**	**16.3**
5. training	**2**	**12.9**
6. management	**15**	**4.7**
7. customer interaction	**4**	**1.2**

*** See section 2.1.**