# Persistent Software Errors

ROBERT L. GLASS

***Abstract*—Persistent software errors—those which are not discovered until late in development, such as when the software becomes operational—are by far the most expensive kind of error. Via analysis of software problem reports, it is discovered that the predominant number of persistent errors in large-scale software efforts are errors of omitted logic..., that is, the code is not as complex as required by the problem to be solved. Peer design and code review, desk checking, and ultrarigorous testing may be the most helpful of the currently available technologies in attacking this problem. New and better methodologies are needed.**

***Index Terms*—Complexity, omitted logic, persistent software error, research in the large, software problem report, testing rigor.**

## Introduction

IT IS well known that software errors vary in expense. That is, software errors which are found quickly and easily, such as syntactic errors and blatantly catastrophic errors, are detected and corrected at little cost (see Fig. 1). On the other hand, those errors which elude normal software review and debug practices, and persist into the software operation/maintenance phase, may be quite expensive.

The expense connected with such errors lies partly in the cost to detect, partly in the cost to correct, and partly in the cost of an inoperable or unsafe software product. Although the first two costs are important, the third is far and away the most significant. Especially in embedded computer systems, such as those controlling aircraft in flight, or a rapid transit vehicle, or a spacecraft, software error cost may be measurable in lives as well as dollars.

Little has appeared in the literature distinguishing between errors by cost. Tools and methodologies for the detection and correction of software errors are proposed and advocated independent of their value in identifying high-expense versus low-expense errors. Software reliability practices and software reliability research which focus on this dichotomy would appear to have large payoff. This paper reports on a study which is an initial effort in that direction.

This study seeks to better understand "persistent" software errors. An error is defined to be persistent if it eludes early detection efforts and does not surface until the software is operational.

Manuscript received July 31, 1979; revised September 2, 1980.
The author is with Boeing Aerospace Company, Seattle, WA 98124.

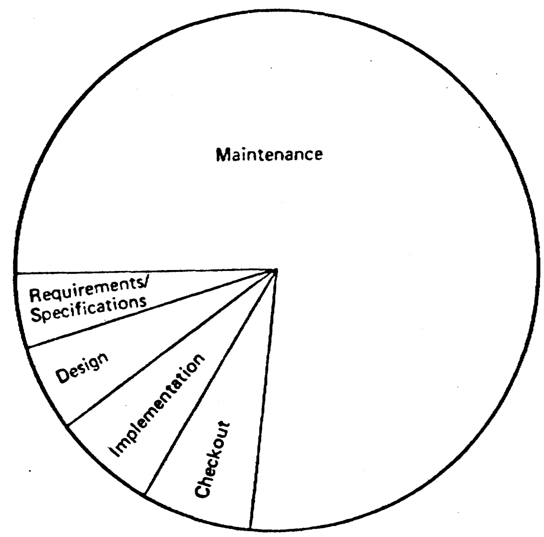



Fig. 1. Software life cycle: per error fix cost per phase.

## The Study

In order to study those kinds of errors, two significant and mature software products were analyzed. Both are operational software systems for military aircraft use. Project A involved 150 programmers at the peak person-load, and contains about a half million instructions in the operational software alone. Project B involved 30 programmers and about 100 000 instructions. Thus, these software products may be considered to be typical of the state-of-the-art in large embedded computer system software.

The size of these software products is important. The point has frequently been made in the literature that large software systems and small software systems are entirely different, and that research "in the small" (using small programs or data/people populations) cannot be extrapolated to be equivalent to research "in the large" [1]-[3], [5], [6]. This study is an example of research in the large; no other approach is likely to be meaningful in the world of large, significant software products.

The method of approach in this study was to examine project-specific software error reports. State-of-the-art methodology in embedded computer systems calls for the filing of a software problem report (SPR) for each software error detected. The report provides spaces for three categories of information: 1) a symptomatic description of the problem from a user point of view, 2) a description of the problem from an internal software point of view, and 3) a description of the software correction. See Figs. 2-4.

0098-5589/81/0300-0162$00.75 

**SOFTWARE PROBLEM REPORT**

SPR No. ______

**PROBLEM: (Prepared by User)**

Originator's Name ______ Organization ______ Phone No. ______

System, Processor, or Component Failing or Project Involved ______ Computer ______ System Version ID ______ Test case or Program ID ______

Description of Problem: The check for a valid platform in task PMSD is illogical. The software checks the data base for non-zero to determine valid platform. The software must expand its logic to determine if the platform-destination combination is valid.

Classification

☒ Error

☐ Information

☐ Revision Request

Correction Required by Date ______ Reference UER No. ______

Authorizing Signature ______ Organization ______ Date ______

**ANALYSIS: (Prepared by organization responsible for software)**

Received Date ______ Time ______

☐ Coding Error

☒ Design Error

☐ Software Not in Error, Explain

☐ Error Previously Reported On SPR No. ______

☐ Others, Explain

Documentation Impact Milestone

1 ☐ 2 ☐ 3 ☐ 4 ☐

5 ☐ 6 ☐ 7 ☐ 8 ☐

Explanation: Insufficient brain power applied during design.

Signature ______ Organization ______ Date ______

**CORRECTION: (Brief description of work performed, including test cases used to confirm correction)**

Solution: Modify code to check if destination is serviced by platform received in input data, or by both platforms.

Run Module Test PM3.1 (completed)

Mod/Programs Changed ______ ______ ______ ______

Hand Load ☐ Yes ☐ No

Work Performed by (Signature) ______ Date ______

**CONFIRMATION: Corrections Verified by Product Assurance**

Signature ______ Date ______

MTM No.(s) ______

Available in (Version ID) ______

J40-122

X 20480 REV 1/75

WHITE - Originator Open GREEN - Analysis CANARY - Originator Closed PINK - Product Control Closed GOLD - Product Control Open

Fig. 2. Omitted logic.

Typically, large software efforts spawn hundreds or even thousands of such reports. SPR's are filed because of real software errors; because of problems which turn out to be errors not caused by software (e.g., computer hardware errors); and for changes which are desired by the user but are not errors. Only the first category of problems—real software errors—was examined in this study.

The SPR's were studied in "raw" (handwritten report) form. Every attempt was made to utilize the information as the programmer reported it, in order to eliminate deletions or transcription error which result from clerical encoding of the information, such as for a computerized database.

The thrust of the study was to divide these SPR's into categories, in order to identify the type of errors which are most

**SOFTWARE PROBLEM REPORT**

SPR No. ____

**PROBLEM: (Prepared by User)**

Originator's Name ____ Organization ____ Phone No. ____

System, Processor, or Component Failing or Project Involved ____ Computer ____ System Version ID ____ Test case or Program ID ____

Description of Problem: Maintenance task MSSW must be queued to operate at the last regular PD on the arrival ramp instead of the first ramp PD.

Classification

☒ Error

☐ Information

☐ Revision Request

Correction Required by Date ____ Reference UER No. ____

Authorizing Signature ____ Organization ____ Date ____

**ANALYSIS: (Prepared by organization responsible for software)**

Received Date ____ Time ____

☒ Coding Error — Explanation: MSSW needs to execute at the last PD where tracking is performed by PPAM to avoid false anomaly reporting by PPAM due to a change in the vehicles assigned point.

☐ Design Error

☐ Software Not in Error, Explain

☐ Error Previously Reported On SPR No. ____

☐ Others, Explain

Documentation Impact Milestone

1 ☐ 2 ☐ 3 ☐ 4 ☐

5 ☐ 6 ☐ 7 ☐ 8 ☐

Signature ____ Organization ____ Date ____

**CORRECTION: (Brief description of work performed, including test cases used to confirm correction)**

Solution: Revise task table data in PSECT SFPDTK to cause MSSW to be queued at PD M32 instead of M30.

Mod/Programs Changed MFSTA

Hand Load ☐ Yes ☐ No

Work Performed by (Signature) ____ Date ____

**CONFIRMATION: Corrections Verified by Product Assurance**

Signature ____ Date ____

MTM No.(s) ____

Available in (Version ID) ____

X 20480 REV 1/75 J40-122

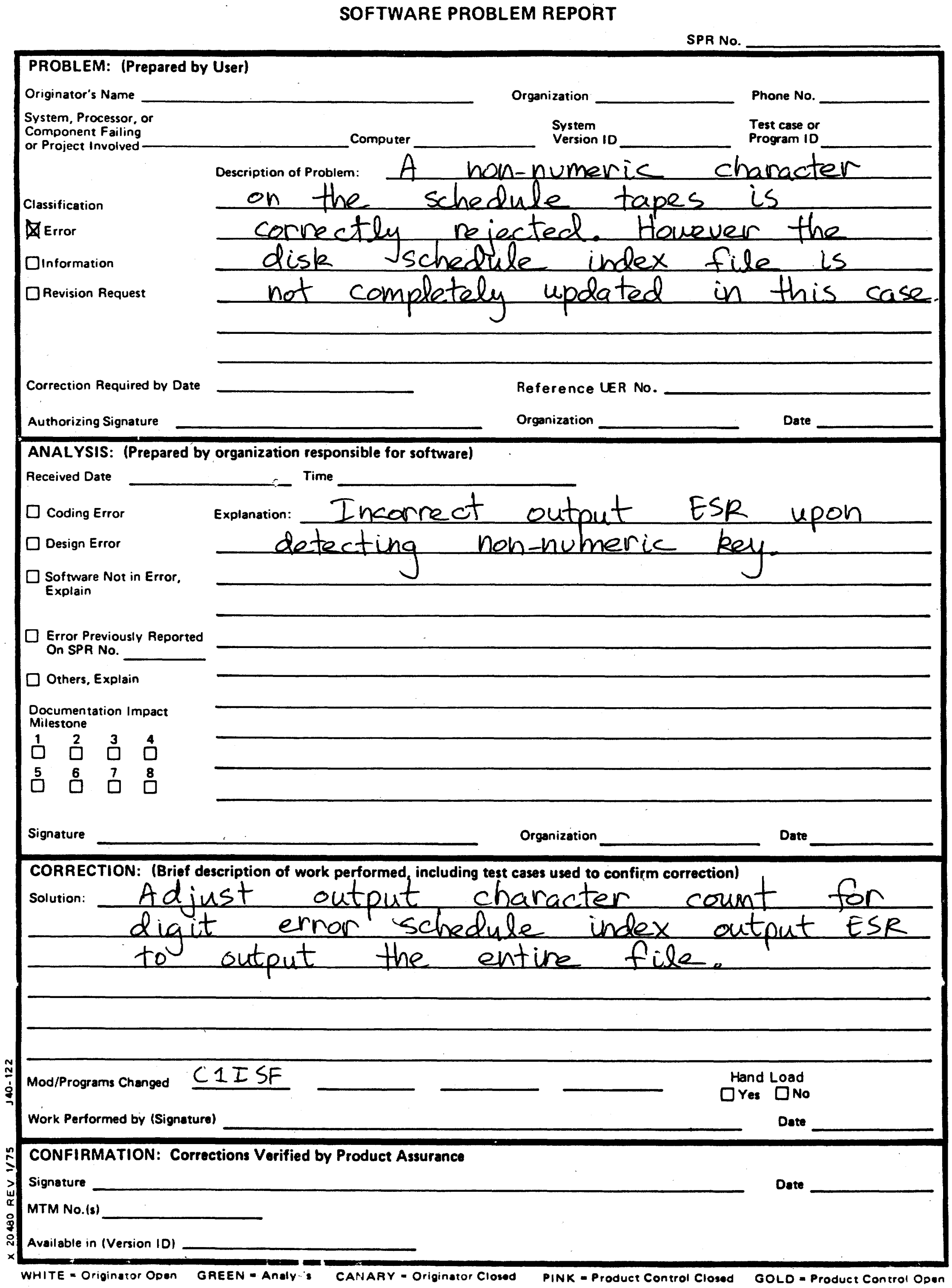
WHITE = Originator Open GREEN = Analysis CANARY = Originator Closed PINK = Product Control Closed GOLD = Product Control Open

Fig. 3. Referenced wrong data variable.

prevalent. Here, an unusual approach was taken. Although error categories are well-known in the literature—TRW developed a pioneering software error category system [10]—those categories were not used in this study. Instead, the errors were allowed to "self-categorize." That is, as each SPR was reviewed, either it was assigned to 1) a category which described its own nature or 2) a category self-generated by some previous error.

There is some controversy attached to the use of raw SPR's and the use of self-categorization.

Regarding raw SPR's, the best approach—that is, the one closest to complete knowledge of the error—would appear to be actual review of the erroneous code and the correction. This has been used by Howden in his error data studies [7]. Use of raw SPR's, however, is an accurate and reproducible approach, since the SPR form repositories are typically sub-



SOFTWARE PROBLEM REPORT

SPR No. _____

PROBLEM: (Prepared by User)

Originator's Name _____ Organization _____ Phone No. _____

System, Processor, or Component Failing or Project Involved _____ Computer _____ System Version ID _____ Test case or Program ID _____

Description of Problem: A non-numeric character on the schedule tapes is correctly rejected. However the disk schedule index file is not completely updated in this case.

Classification

☒ Error

☐ Information

☐ Revision Request

Correction Required by Date _____ Reference UER No. _____

Authorizing Signature _____ Organization _____ Date _____

ANALYSIS: (Prepared by organization responsible for software)

Received Date _____ Time _____

☐ Coding Error

☐ Design Error

☐ Software Not in Error, Explain

☐ Error Previously Reported On SPR No. _____

☐ Others, Explain

Explanation: Incorrect output ESR upon detecting non-numeric key.

Documentation Impact Milestone

1 ☐ 2 ☐ 3 ☐ 4 ☐

5 ☐ 6 ☐ 7 ☐ 8 ☐

Signature _____ Organization _____ Date _____

CORRECTION: (Brief description of work performed, including test cases used to confirm correction)

Solution: Adjust output character count for digit error schedule index output ESR to output the entire file.

Mod/Programs Changed C1ISF _____ _____ _____ Hand Load ☐ Yes ☐ No

Work Performed by (Signature) _____ Date _____

CONFIRMATION: Corrections Verified by Product Assurance

Signature _____ Date _____

MTM No.(s) _____

Available in (Version ID) _____

X 20480 REV 1/75 J40-122

WHITE = Originator Open GREEN = Analy s CANARY = Originator Closed PINK = Product Control Closed GOLD = Product Control Open

Fig. 4. Omitted logic.

ject to configuration management practices. The decision to use raw SPR's in this case was purely pragmatic—a large number of errors can be reviewed rapidly with minimal loss of authenticity.

Regarding self-categorization, the study was built on the premise of exploring new ground. That is, to the author's knowledge, no one has studied persistent software errors per se before. To avoid the possibility that traditional error categories were not appropriate to persistent errors, the traditional categories were deliberately avoided. Self-categorization, of course, has the flaw that the judgment of the individual researcher will have some impact on the final results. However, the traditional categories discussed in [10] are ambiguous enough that they share this problem.

In any case, 100 software errors from each of two projects were subjected to review at the raw SPR level via self-

TABLE I
PERSISTENT SOFTWARE ERRORS BY FREQUENCY OCCURRENCE (200 ERRORS EXAMINED, 100 ON EACH PROJECT)

| Category: | Project A | Project B | Total: |
|---|---|---|---|
| 1. Omitted logic (existing code too simple) | 36 | 24 | 60 |
| 2. Failure to reset data | 17 | 6 | 23 |
| 3. Regression error | 5 | 12 | 17 |
| 4. Documentation in error (software correct) | 10 | 6 | 16 |
| 5. Requirements inadequate | 10 | 1 | 11 |
| 6. Patch in error | 0 | 11 | 11 |
| 7. Commentary in error | 0 | 11 | 11 |
| 8. IF statement too simple | 9 | 2 | 11 |
| 9. Referenced wrong data variable | 6 | 4 | 10 |
| 10. Data alignment error (leftmost vs. rightmost bits, etc.) | 4 | 3 | 7 |
| 11. Timing error causes data loss | 3 | 3 | 6 |
| 12. Failure to initialize data | 4 | 1 | 5 |
| 13. Other categories of lesser importance (total 4 or less) - Logic too complex, compiler error, data storage overflow, expression incorrectly coded, pointer one off, dynamic allocation failure, data not included in checkpoint, microcode error, data boundary problem, macro error, multitasking synchronizing error, erroneous initialization, naming conventions violated, logic order incorrect, interface mismatch, data reset in error, parameter mismatch, inefficient code, data declaration wrong, bad overlay, statement label at wrong place, data clobbered. | | | |

NOTE:
An error was allowed to tally in more than one category. "Failure to reset data", for example, is almost always a specific instance of "omitted logic." So are "if statement too simple" and "failure to initialize data." Any error could also be a "regression error."

categorization techniques. The errors were considered to be persistent on the basis that they were the most recent errors detected on those production-status projects. In most cases, this proved to be a sufficiently valid basis. In some cases, however, the errors were spawned by the correction of other persistent errors (these are commonly called "regression errors"). Errors of this class were included in the study on the grounds that such regression errors are just as costly as nonregression persistent errors; however, these errors were allowed to self-categorize a category for themselves.

An error was allowed to tally in more than one category. No attempt was made to provide mutually exclusive categories; the emphasis was on creating a realistic summary of the data as it was analyzed, and not to force it to fit an externally applied artifice. For example, the SPR in Fig. 2 would have been categorized both as "omitted logic" and "if statement too simple."

The process of analysis, then, was simply this: 1) project-specific, configuration-managed SPR forms, filed in chronological sequence, were examined one at a time; 2) using the "problem," "analysis," and "correction" information (see Figs. 2-4) the nature of the error was ascertained; 3) that error was categorized into its own and/or a previously selected category; 4) a tally was added to those categories. At the conclusion of analysis of a set of project-specific SPR's, tallies for the (variable number of) categories were summed. The categories for one project overlapped partially but not entirely with those of the other project (e.g., in Table I note that project A had no patching or commentary errors. Presumably project A did not allow patches and did not file SPR's on commentary errors).

## THE FINDINGS

The findings of this study appear to be significant. That is, the categorized persistent SPR's show a consistent and definite pattern (see Table I).

The major finding of the study is that a large percentage of persistent software errors are instances of the software not being sufficiently complex to match the problem being solved. It is as if the programmer mind is straining to handle the complex interrelationships of a problem solution, and has failed. For example, a large number of such errors are the result of a predicate not having enough conditions—some flag or piece of data was not taken into account when it should have been—or of a variable not being reset to some baseline value after a major functional logic segment has finished dealing with it.

Here again, it is important to distinguish between the large problem and the small problem environment. Intuitively, it is easily seen that this kind of error is much more likely to emerge in the large rather than the small problem. The interrelationships between data and logic are much more entwined and complex in the large problem environment. It has even been said that "a 25 percent increase in problem complexity leads to a 100 percent increase in program complexity" [11]. And, in fact, most professional programmers have built software which they then realized was, in some areas, beyond their ability to comprehend (in the sense that its results were not predictable prior to its execution).

These problems of complexity may be considered to be design errors, and indeed many of them are. It is well known that design errors dominate the population of software errors (e.g., [4]). However, it must be recognized that they are the kind of design error which occurs at the most detailed level,

TABLE II
ERROR CATEGORY DEFINITION

| Category: | Definition, Example: |
|---|---|
| 1. Omitted logic | Code is lacking which should be present. Variable A is assigned a new value in logic path X but is not reset to the value required prior to entering path Y. |
| 2. Failure to reset data | Reassignment of needed value to a variable omitted. See example for "omitted logic." |
| 3. Regression error | Attempt to correct one error causes another. |
| 4. Documentation in error | Software and documentation conflict; software is correct. User manual says to input a value in inches, but program consistently assumes the value is in centimeters. |
| 5. Requirements inadequate | Specification of the problem insufficient to define the desired solution. See Figure 4. If the requirements failed to note the interrelationship of the validity check and the disk schedule index, then this would also be a requirements error. |
| 6. Patch in error | Temporary machine code change contains an error. Source code is correct, but "jump to 14000" should have been "jump to 14004." |
| 7. Commentary in error | Source code comment is incorrect. Program says DO I=1,5 while comment says "loop 4 times." |
| 8. IF statement too simple | Not all conditions necessary for an IF statement are present. IF A<B should be IF A<B AND B<C. |
| 9. Referenced wrong data variable | Self-explanatory See Figure 3. The wrong queues were referenced. |
| 10. Data alignment error | Data accessed is not the same as data desired due to using wrong set of bits. Leftmost instead of rightmost substring of bits used from a data structure. |
| 11. Timing error causes data loss | Shared data changed by a process at an unexpected time. Parallel task B changes XYZ just before task A used it. |
| 12. Failure to initialize data | Non-preset data is referenced before a value is assigned. |

Lesser categories are not defined here.

where the design blends into its code. Since at this level it is not clear what is design and what is code, it would be simplistic to attach these errors to the broader category "design error."

## WHAT TO DO ABOUT THE FINDINGS

The findings of this study are unsettling. They are unsettling not because they are a refocusing of our understandings of software problem solutions. They are unsettling because it is not at all clear what we should do about them.

Philosophically, what is needed is obvious. We need a human mind extender, one which makes it possible for the human mind to conceive problems and solutions beyond its current capacity. (Note that the analysis section of Fig. 2 says "insufficient brain power applied during design")!

That, of course, is naive, given the current state-of-the-art. And yet, such a solution can at least be considered.

How could the mind be extended in those specific directions? Perhaps by very high-order languages, which remove solution details from the domain of the programmer into the domain of the compiler (analogous to computer hardware register management being moved into the high-order language compiler)?

Perhaps by a design aid which manages and analyzes design details which the human mind cannot?

Perhaps by a maintenance tool which extracts from existing software its underlying design elements, and subjects them to (human-assisted?) consistency analysis?

Those answers are not very satisfying, for they are beyond the state-of-the-computer-art. And yet they are promising, because they represent a level of computer application breakthrough which, if achieved, obviously transcends the software engineering problem which spawned it.

Of course, there are mundane but useful answers. If the designer, the implementer, and the tester employ a deep level of concentration and rigor, the omitted logic error is preventable or detectable. In-depth technical peer design reviews and peer code reviews can, for example, detect these errors before they become "persistent." Rigorous test case definition, especially where the test cases are driven by comprehensive specifications, can also detect most "persistent" errors early. Simple traditional desk checking, if properly applied, can also do the job.

The problem with these mundane but useful solutions is that, in the complex problems being solved by today's professional programmer, the necessary concentration and rigor is difficult to achieve. Still, given proper application they can be viable solutions.

## CONCLUSION

Persistent software errors are seen to be dominated by a class of error which can be categorized as "the failure of the problem solution to match the complexity of the problem to be solved." Examples of such errors are predicates with insufficient conditions, and failure to reset data to some baseline value after its use in a functional logic segment.

The solution to this class of problems is difficult. Somehow, the programmer's mind must be extended to encompass complexity beyond its current capability. This is obviously a solution beyond the current state-of-the-art.

Solutions which can be effective for today's large software system producer are maintaining awareness of the problem, and spending more time analyzing complex interrelationships via peer review, program desk checking, and rigorous testing. As is already well known, identifying those (persistent) problems early in the software life cycle can have major positive cost impact on total system cost.

## ACKNOWLEDGMENT

The ideas and support of D. Feinberg, L. MacLaren, R. Noiseux, and E. Presson have been important in the development of this research.

## REFERENCES

[1] F. P. Brooks, *The Mythical Man-Month*. Reading, MA: Addison-Wesley, 1975.

[2] J. R. Brown, "Impact of MPP on system development," RADC-TR-77-121, 1977.

[3] F. DeRemer and H. H. Kron, "Programming-in-the-large versus programming-in-the-small," *IEEE Trans. Software Eng.*, vol. SE-2, pp. 80–86, June 1976.

[4] A. B. Endres, "An analysis of errors and their causes in system programs," *IEEE Trans. Software Eng.*, vol. SE-1, pp. 140–149, June 1975.

[5] R. L. Glass, "Small versus large projects," in *Software Reliability Guidebook*. Englewood Cliffs, NJ: Prentice-Hall, 1979, pp. 21–25.

[6] E. Horowitz, *Practical Strategies for Developing Large Software Systems*. Reading, MA: Addison-Wesley, 1975.

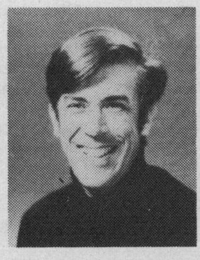
[7] W. E. Howden, "An analysis of software validation techniques for scientific programs," Univ. Victoria Rep. DM-171-1R, 1979.

[8] J. R. Stanfield and A. M. Skrukrud, "Software acquisition management guidebook–Software maintenance volume," Syst. Develop. Corp., TM-5772/004/02, Nov. 1977.

[9] N. F. Schneidewind and H. M. Hoffman, "An experiment in software error data collection and analysis," *IEEE Trans. Software Eng.*, vol. SE-5, pp. 276–286, May 1979.

[10] A. N. Sukert, "A multi-project comparison of software reliability models," in *Proc. AIAA Conf. Comput. in Aerospace*, 1977.

[11] S. N. Woodfield, "An experiment on unit increase in program complexity," *IEEE Trans. Software Eng.*, vol. SE-5, pp. 76–79, Mar. 1979.



**Robert L. Glass** received the B.A. degree in mathematics from Culver-Stockton College, Canton, MO, in 1952, and the M.S. degree in mathematics from the University of Wisconsin in 1954.

He is presently a Senior Computing Specialist with Boeing Computer Services, a member of a team developing a UCSD Pascal based microcomputer system. Prior to that, he had spent 25 years in Aerospace computing–3 years with North American Aviation, 8 with Aerojet-General, and 14 with Boeing Aerospace. He has been an active participant in the Ada language and environment definition process, and in the JOUIAL language User's Group. He is experienced in scientific and commercial applications programming, and especially in the construction of system software. He is the author of three software engineering books–*Software Reliability Guidebook* (1979), *Software Maintenance Guidebook* (1981), and *Software Soliloquies* (1981)–and of four humorous computing books, two about computing projects which failed, and two about computing people. He has been published in several IEEE and ACM publications, and is a frequent contributor to *Computerworld* and *Datamation*. He has been an ACM National Lecturer for four years. His outlook is that of the experienced and skilled software craftsman, not that of a manager or an academician.