Prediction of Software Failures

Myron Lipow

Software Product Assurance, TRW Defense and Space Systems Group, Redondo Beach, California

Reduction in computer software errors by use of preventive and detective tools and techniques is illustrated using data from three large software projects. A trade-off analysis of costs of early application (during design) of a tool to prevent errors with costs of later correction (during test) is presented using a simplified model, and also illustrated using data from one of the projects.

INTRODUCTION

Errors are made in computer software at all stages of the production process. There is some evidence that a substantial majority of errors is committed during the requirements and design phase (over 60%) and less than 40% during coding. Delay in detection and correction of errors until the testing phase will result in substantially higher costs than if the errors are discovered earlier. Consequently, inexpensive means of detecting and preventing errors applied during requirements analysis and design could significantly reduce costs.

Selection of the best preventive and detective techniques and tools depends upon how thoroughly error data collection is done and how errors are categorized. Data from three large software projects are examined and an analysis of the effect on errors of certain preventive and detective tools and techniques is presented. Also, a trade-off analysis of costs of early application of an error-preventing tool vs costs of later correction will be illustrated.

DEFINITIONS OF TERMS RELATED TO ERRORS

In order to be able to analyze errors and thereby determine the most appropriate techniques or tools for preventing or detecting them, some precise definitions of terms related to errors are necessary, as given in Table 1. Some of the definitions are controversial and not uniformly accepted, but similar interpretations of the terms have been developed by the Terminology Task Group of the IEEE Technical Committee on Software Engineering and are presently under review.

The major points of Table 1 are that *errors* are considered to be mistakes committed by people, including those involved in formulating requirements, in design, and in producing code. *Faults* are the visible or invisible evidence of the error. They reside in the code or documentation. A fault may not always result in a failure attributable to the software, even if the code containing the fault is executed, depending upon the input data.

For completeness, and perhaps further controversy, definitions of software reliability from two viewpoints, those of the user and the developer, are given. The latter will not be satisfied if he knows the software to contain faults, and considers the software to have a reliability of zero unless the faults are removed. The former will tolerate software failures from time to time, provided that his system is nearly always "up" and that the failure is not so critical as to affect performance of most of the important functions while corrections are being made.

COSTS OF ERRORS

Discovering errors early in software production can yield a large payoff [1]. There it is shown that if cost of an error discovered during coding is set at a unit value, it is still between five and ten times the cost of correcting the error if its cause were found during the requirements specification phase. Toward the end of development, during acceptance tests, the relative cost of correcting an error discovered then is about 5; during operations, the relative cost may be as

Address correspondence to Myron Lipow, Software Product Assurance, TRW Defense and Space Systems Group, Mail Stop 88/1011, One Space Park, Redondo Beach, California 90278.

© Elsevier North Holland, Inc., 1979
0164-1212/79/010071-05$02.25

Table 1. Definitions

Error: (causative)
  A conceptual, syntactic, or clerical discrepancy which results in one or more faults in the software.
Fault: (symptomatic)
  A specific manifestation of an error. A discrepancy in the software which impairs its ability to function as intended. An error may be the cause of several faults.
Failure:
  A software failure occurs when a fault in the computer program is evoked by some input data, resulting in the computer program not correctly computing the required function.
Software:
  Computer program products in the form of card decks, magnetic tapes, or disks, and all descriptive documentation, including specifications, listings, manuals, and flow charts.
Software Reliability:
  System, user, or "macroscopic" viewpoint:
    Probability that use of the software does not result in failure of the system to perform as expected by more than a tolerable frequency.
  Subsystem, developer, or "microscopic" viewpoint:
    Probability that the program, routine, or module is fault-free.

Table 2. Some Types of Faults in Requirements Specifications

Decision criteria, accuracy criteria, processing rates, error recovery requirements missing, incomplete or inadequately stated.
Incorrect requirements
  Model does not fit physical situation well enough
  Document references incorrect
Inconsistent or incompatible requirements
  Two locations in specification give conflicting information
  Conventions (e.g., coordinate systems) not consistent with other documentation
Requirement unclear or nonsense
Requirement left out of specification document

much as 20.[1] Comparing extreme ends of the software life cycle, therefore, we find that 1 man-hour spent in finding and correcting a requirements specification error during the requirements phase could be multiplied by as much as 100 were the error to be discovered during user operations.[2] The additional costs as errors are discovered downstream of their origination are primarily due to the greater difficulty of determining the correction, including the retest, reanalysis, and documentation changes necessary.

REQUIREMENTS ERRORS

The discipline of *software requirements* engineering has shown a rapid growth due to the recognition that deficiencies in the specification that describes what the software will do become more difficult and expensive to correct later.

In recent studies [3] evidence has shown that over 60% of the errors occur during the requirements formulation, preliminary design, and detailed program design phases of software development, and less than 40% arise in the programming or coding phase. Furthermore, many errors due to incorrect, inconsistent, incomplete, or misunderstood requirements statements are detected, but not necessarily resolved, during subsequent preliminary and detailed design phases. The unresolved deficiencies may, in fact, remain open to the point when the user finds that the software does not perform satisfactorily for his needs during the operations phase, owing to previous inability to ascertain or express these requirements adequately. This can occur particularly in embedded systems when hardware is unable to meet its requirements, resulting in unforeseen changes to software requirements at a time far removed from the design phase.

Table 2 describes some of the faults that are found in requirements specifications. A series of papers on methodologies that are being developed to prevent most of these problems has appeared elsewhere [4].

DESIGN ERRORS

What are the causes of errors in design? Answers are elusive because this phase of software development involves an intellectual, creative, or intuitive process, often highly dependent upon the personality of the designer. The best answer perhaps is that many errors can be prevented by consideration and avoidance of the following "poor" approaches:

1. *Inadequate simulation.* There are very few aids to help the designer to make overall software/hardware trade-off decisions in order to narrow the number of degrees of freedom available. One published technique is based on the *Extendable Computer System Simulator* [5] making it possible to develop a functional simulation of a system in a much shorter time than it takes to develop the complete design itself.
2. *Deficient design representations.* Flow charts, the primary method, are too easy to construct in a complicated manner, and hard to understand and maintain. Machine processable structured de-

[1]This number is probably an extreme upper limit [2].
[2]During the operations and maintenance phase itself, costs of correcting errors arising because of new or changed requirements would similarly expand during this change process, but by less than 100 to 1, unless memory usage is approaching (say) 95% of that available.

sign languages ("structured pidgin English") are probably more suitable. Program Design Language [6] is probably the best example of such a design aid.

3. *Unstructuredness.* Structuredness is a general term referring to a design philosophy requiring adherence to a set of rules or enforced standards that embodies such techniques as *top-down design, program modularization,* or *independence, structured control flow,* and so forth. Excellent detailed discussions on the topic of structuredness in computer program design are extant [7,8,9].
4. *Use of unstandardized languages.* Standardization implies, among other requirements, that rigid configuration controls be kept on compilers, support tools, and documentation. The U.S. Department of Defense, having recognized language standardization as an important factor in production of reliable software, published Department of Defense Directive (DODD) 5000.29 in 1976, which required use of High-Order Languages except when an assembly language could be justified. Following this directive, DOD Instruction 5000.31 was issued, which specified several "approved" languages.

The growing complexity of software/hardware interfaces as a design factor in software also cannot be neglected. Choice of hardware alternatives has been increasing at an enormous rate. As compared to the 1950s, the likely number of choices for central processor units or peripherals in the 1970s is about 100 to 1 and will be even higher in the 1980s [1].

CODING ERRORS

Errors in coding occur primarily in the following ways:

1. *Typographical errors.* A programmer incorrectly writes down or copies a statement in the source language; e.g., omitting parentheses, writing down "23" instead of "32," or misspelling a variable name.
2. *Misinterpretation of language constructions.* A programmer uses certain language constructions in a way he or she believes is correct, but the compiler interprets them differently.
3. *Missing or incorrect logic.* Assuming that the design specification was correct, a programmer makes an error by omitting a required test for a condition, or (say) incorrectly tests for a condition after a scope is executed, whereas if the condition were false, the scope should not have been executed.
4. *Undocumented assumption.* The programmer makes an error by assuming a design interpretation without telling anyone, when actually the design was ambiguous and allowed two or more interpretations. This is a gray area, since responsibility for the error could be placed entirely on the designer. [For example, the programmer assumes that the first quadrant in the (x,y) plane is defined by $(x > 0, y > 0)$ whereas the designer should have specified it as $(x \geq 0, y \geq 0)$, according to the conditions of the problem.]
5. *Singularities and critical values.* The programmer forgets to test and provide responses to division by zero, or singular points or regions of library functions (square roots or logarithms of negative numbers).
6. *Algorithm approximation.* A programmer may use an approximation to the correct equations to make their solution tractable, or to a function in order to increase execution speed. The approximations may be insufficiently accurate over the required ranges of the variables. Again, responsibility for the error could be placed upon the designer in most cases.
7. *Data structure defects.* The program is incompatible with the data structure specification; e.g., a table may be specified to contain a maximum number of entries but the program may continue to try to insert entries into the table when it is already full.

The types of coding error expressed above are almost never detected during compilation, and perhaps not for many test runs or executions of the program, or not at all. There are, fortunately, a growing list of software tools or automated aids that can find many of these errors at reasonable cost.

FAULT CATEGORY ANALYSIS

Data given in Table 3 are based upon a careful analysis of three major projects [3] and show percentages of faults by major category occurring during

Table 3. Fault Categories (Symptomatic) and Frequencies

Logic	26%
Data handling	18%
Interface	16%
Data input/output	14%
Computational	9%
Data base	7%
Data definition	3%
Other	7%

Table 4. Example of Impact[a] of Fault-Preventive Techniques and Tools

	Preventive technique		Preventive tool
Major fault category	Design inspection	Code inspection	Design language
Logic	23%	19%	15%
Data handling	13%	13%	5%
Interface	9%	8%	5%
Data input/output	5%	13%	5%
Computational	6%	4%	3%
Data definition	1%	1%	0%
Data base	0%	2%	0%
Other	1%	2%	0%
Totals	58%	62%	32%

[a]In percentages of faults susceptible to prevention by the given technique or tool. Thus, of the 26% that are logic faults in Table 3, 23/26 or 88% would be susceptible to design inspection.

the requirements, design, and coding phases, and discovered during testing. The data represent averages weighted by the numbers of test problem reports for each project.

IMPACT OF TECHNIQUES AND TOOLS

Table 4 presents the results of an analysis of the potential impact of some preventive techniques (design inspection and code inspection) and a preventive tool (a design language) on the errors of one of the projects included in the data of Table 3. Similarly, Table 5 presents an analysis of the potential impact of fault detection techniques and a detection tool (code standards auditor) on the errors of the same project. Other techniques and tools were considered also, and the most effective of each group are shown in Tables 4 and 5. Although many different tools may cumulatively prevent or detect a larger fraction of faults than that shown in the tables, it may be that the nonautomated methods we know of may do their job better or more completely. The question of cost effectiveness must be considered here, however, since a tool once developed will probably be much cheaper to use than a manual, though possibly more effective, technique. The next section examines in a general way the trade-off of costs of a new tool for detecting requirements or design errors prior to coding vs the costs of correcting the errors during test.

Table 5. Example of Impact[a] of Fault Detection Techniques and Tools

	Detection technique	Detection tool
Major fault category	Path execution, data singularity, and extremes test	Code standards auditor
Logic	21%	2%
Data handling	14%	8%
Interface	7%	4%
Data input/output	16%	3%
Computational	8%	1%
Data base	4%	0%
Data definition	1%	1%
Other	2%	0%
Totals	73%	19%

[a]In percentages of faults susceptible to detection by the given technique or tool.

COST TRADE-OFF OF TOOL VS LATER COSTS OF CORRECTING ERRORS

A simple cost model that considers the use of a tool or technique to detect additional errors during the design phase, and thereby save some of the greater expense of correcting the errors during the test phase, is discussed in the following paragraphs.

The following cost factors are defined:

C_D is the cost per error of correcting a design error[3] during the design phase.

C_T is the cost per error of correcting a design error discovered during the test phase.

C_0 is the cost of developing and applying a tool or technique that detects an additional fraction P of design errors during the design phase.

Also,

N_D is the number of design errors discovered and corrected in design review during the design phase. (We assume, for simplicity, that each error discovered is permanently corrected.)

N_T is the number of design errors discovered and corrected during the test phase.

M_T is the total number of errors (design plus coding) discovered during test.

The total cost of discovering and correcting design errors during design and test phases without the new tool or technique is

$$C_1 = C_D N_D + C_T N_T.$$

The total number of design errors discovered and corrected during the design phase using the new tool or technique is $N_D(1 + P)$. Consequently the number of design errors discovered and corrected during the

[3]The term "design error" as used here includes both errors made in the design phase and errors made in specifying requirements.

test phase would then be reduced to $\sim N_T(1 - P)$. Therefore, had the tool or technique been used, the cost of discovering and correcting design errors during design and test, plus the cost of the new tool or technique, would be

$$C_2 = C_0 + C_D N_D(1 + P) + C_T N_T(1 - P).$$

The tool or technique will pay for itself if

$$C_1 - C_2 > 0$$

or if

$$C_0 < P(C_T N_T - C_D N_D).$$

Example. We use data from one of the software projects analyzed in Ref. 3, and some assumptions. First, Ref. 1 indicates that the average cost of correcting an error during the test phase is about 10 times the cost of correcting it during the design phase. Thus $C_T \simeq 10\ C_D$. For this particular project the total number of errors discovered during test M_T was about 24% of the number of errors discovered and corrected in design review during the design phase N_D; i.e., $M_T \simeq 0.24\ N_D$. Furthermore, the number of design errors discovered during test N_T was estimated to be about 62% of M_T (i.e., about 38% of the total number of test errors were made when producing code). Thus $N_T \simeq 0.62\ M_T = (0.62)(0.24)\ N_D \simeq 0.15\ N_D$. For an estimate of P, from Tables 4 and 5 it appears that $P \simeq 0.5$ may be appropriate. Consequently, using the information that has been derived so far, the last inequality becomes

$$C_0 < 0.25\ C_D N_D.$$

Additionally, we know that in the particular project there were $N_D \sim 5400$ design problem reports. Also a reasonable guess is that $C_D = \$25$/error. Thus, if

$$C_0 < \$33,750$$

is satisfied, then the tool or technique would be worthwhile.

CONCLUSIONS

Some of the conclusions reached within this paper are only probable hypotheses and should be tested with other software data. In summary they are as follows:

1. Based upon the observation that more errors originate during requirements and design phases than during the coding phase, more effort needs to be spent in requirements and design validation.
2. Existing tools may be less effective than practices and procedures for preventing or detecting faults.
3. Use of tools or techniques in early requirements or design phases to find errors at that time may be cost effective if the cost of developing the tool or technique including application is low enough.

REFERENCES

1. B. W. Boehm, "Software Engineering," *IEEE Trans. Comput.* C-25 (12), 1226–1241 (1976). Also published in the TRW Software Series TRW-SS-76-08, October 1976.
2. R. W. Wolverton, private communication, 1978.
3. T. A. Thayer, M. Lipow, and E. C. Nelson, *Software Reliability—A Study of Large Project Reality*, TRW Series on Software Technology, Vol. 2, Elsevier North Holland, 1978.
4. *IEEE Trans. Software Eng.* SE-3 (1) 2–69 (1977) (Special collection on requirements analysis).
5. D. W. Kosy, The ECSS II Language for Simulating Computer Systems, The Rand Corporation, Report No. R-1895-6SA, December 1975.
6. Caine, Farber & Gordon, Inc., *Program Design Language*, Pasadena, California, 1974, 1975.
7. R. C. Tausworthe, *Standardized Development of Computer Software, Part I, Methods*, Jet Propulsion Laboratory, Calfornia Institute of Technology, Pasadena, California, 1976.
8. E. Yourdon, *Techniques of Program Structure and Design*, Prentice-Hall, Englewood Cliffs, New Jersey, 1975.
9. RADC Technical Reports TR-74-300, Vols. I–XV, Structured Programing Series, IBM, 1974–1976.