Understanding software through empirical reliability analysis*

by T. A. THAYER
TRW Systems Group
Redondo Beach, California

INTRODUCTION

The need for improving the reliability of delivered software is becoming increasingly obvious to both the purchasers and producers of today’s software systems. As noted by Boehm,¹ the records show many examples of software systems which, when delivered for operational use, either performed in a degraded fashion or failed to perform at all. The results are higher software costs and delays in operational usage.

Purchasers of software are demanding more reliability but are not presently sure how to specify what they want in the form of written requirements. Producers of software, in an effort to provide this increased reliability, are imposing new controls on the software development process. These new controls are both managerial (chief programmer teams, independent test and audit groups, etc.) and technological in nature (structured programming, coding standards, test tools, etc.). Although these controls are intuitively beneficial, little hard data exist to support claims of benefit. There is even less information on the benefits of these controls in specific software applications; e.g., batch versus real time software, OS versus applications software, programs written in assembly language versus higher order language.

Assuming that information is collected and available, what can we expect to gain from its analysis? The long range benefits are the most obvious. We will learn, for example, how large a software module can be without affecting understandability and how small it can be before partitioning problems are encountered. Specific tools and techniques to improve the development process will also be identified. These are all things that can be applied in the future, to the next project. But, what about the near term payoff for the project supplying the data? A project manager, when asked to contribute to data collection and analysis activities, will invariably ask if there will be a benefit to his project. The benefit to the on-going project comes from increased awareness of problems and better control over the development process, both contributory to production of more reliable software. Adequate data on software reliability can thus aid both current and future development projects—if methods for analyzing and understanding these data are developed.

In a study* being performed by TRW for the Rome Air Development Center, data from four large software systems are being analyzed to determine the types of errors found in software during testing. The objective is principally to recommend new development or test techniques for the detection and prevention of software errors, but we are also attempting to model software reliability. In the course of supplying real** data descriptive of software reliability and for model evaluation, we have had to determine (1) what data are generally available, (2) methods for collecting and storing these data, (3) methods for describing software errors, (4) methods for characterizing the software, and the development and test processes in quantitative terms, and finally (5) methods of analysis. Although the projects studied have varied greatly in size, language, operating mode, and structure, the data available during the development process were similar for each project: error data, recorded in various forms of software problem reports (SPR) and ancillary project data needed to understand and support analysis of the error data. Although the data were not generated specifically for the study, we found that we could do much to quantify software reliability and the characteristics of the software itself, as well as improving our understanding of both the software and the development process. Some results of the Software Reliability Study will be presented to illustrate the benefits of software reliability data collection and analysis. Also presented are some recommendations for identifying data that need collecting.

SOFTWARE RELIABILITY

As the reader will note, the term “software reliability” has been used several times in the foregoing paragraphs without definition. Although the term is defined in a number of references,¹⁻³ generally in conjunction with modeling work, the following definition is taken and offered as a background for the empirical approach to characterizing reliable software:

Software possesses reliability to the extent that it can be expected to perform its intended functions satisfactorily.¹

---

* Portions of the work presented in this paper are sponsored by Rome Air Development Center under Air Force Contract F30602-74-C-0036.

** The Software Reliability Study.

[From the collection of the Computer History Museum (www.computerhistory.org)]
Assuming that documented errors represent an inability to perform intended functions satisfactorily, error-free software would be reliable software. This assumption is one that is made here.

In the definition above, the word "satisfactorily" also begs definition. A universally applicable quantitative definition does not presently exist; however, in the world of software problem reports mere creation of the report registers some dissatisfaction with performance, whether the report documents a real problem, a need for software enhancement, or what turns out to be no problem at all.

Using the definition given above, the number of errors documented on problem reports can be used as an indicator of software reliability. However, raw counts of errors don’t tell the whole story and can be misleading. Therefore, further analysis is necessary to determine the types of errors, when and where they were introduced, how they were detected, and their impact on the operation of the software system.

It is possible to categorize software problem reports symptomatically, at the user level, and according to cause, at the code level. Symptomatic categories describe the failure, and causative categories describe the error. These are often quite different. Attempts to establish software error category lists for multi-project applications have met with some success.

The principal findings being that errors are repeated and tend to fall into common categories, that lists have to provide a fair amount of detail to yield useful information, and that they also have to be short and comprehensive so that the fixer* of the problem can be encouraged to provide the analysis. A portion of an error category list is given in Figure 1 along with empirical data from three software projects. This list presents data generated during development of a new software system (Project 3), during development of four updates to an operational system (Project 2), and during operational usage (Project 4).

SOFTWARE CHARACTERIZATION

In order to better understand why elements of the software have high or low error frequencies, it is necessary to collect data on the software itself. Two types of software characteristics were defined in the Software Reliability Study, structural and subjective characteristics. Structural characteristics are those that can be obtained from the source code, where the errors are ultimately found. These character-

* He alone is close enough to the actual error to assign a causative error category and he must make the assignment while the solution is fresh in his mind.

Figure 1—Sample error category list
were five routine types* considered, too: executive or code, debug/test, implementation, and documentation. Difficulty and software type were difficulty and software type. Difficulty was assigned to each of five development disciplines: design, control, setup, computational, data handling, and input or output. These data were very helpful in understanding error histories.

Subjective characteristics are those that are obtained from the people who produced the software; the programmers, the design engineers, the testers, and their managers. Although methods for quantifying these subjective characteristics are not well established, this information can add insight to the error history data that is not evidenced by the more accurately measured structural characteristics. However simple a routine might be structurally, if it was *fold* that the routine was (or would be) a problem for some other reason, the error history can be affected by this feeling. A typical example concerns the software that has been produced previously for another project. The initial feeling is that it should be an easy task to convert an existing routine for use in a similar (but not identical) software system. This situation invariably receives less attention than the newer software and just as invariably represents more work than was originally conceived. On the other hand, portions of the software thought to represent potential problems are given much attention, and this can be seen in the error history data.

Subjective characteristics dealt with in the Software Reliability Study were difficulty and software type. Difficulty was assigned to each of five development disciplines: design, code, debug/test, implementation, and documentation. There were five routine types* considered, too: executive or control, setup, computational, data handling, and input or output. These data were very helpful in understanding error histories.

*Note that these metrics are, by design, relatively simple and incorporate parameters which are thought to be available from almost any software system.

**This particular form of the equation resulted from a combination of intuition about loop complexity and evaluation with real data where nesting reached the tenth level in some instances.

<table>
<thead>
<tr>
<th>TABLE I—Routine Structural Characteristics</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Routine size (source statements and machine instructions)</td>
</tr>
<tr>
<td>• Number of branches</td>
</tr>
<tr>
<td>• Branch statements and nesting levels</td>
</tr>
<tr>
<td>• Entry/exit points</td>
</tr>
<tr>
<td>• Routine interfaces</td>
</tr>
<tr>
<td>• Data base interfaces</td>
</tr>
<tr>
<td>• Calling sequence arguments</td>
</tr>
<tr>
<td>• Code type</td>
</tr>
<tr>
<td>—computational</td>
</tr>
<tr>
<td>—data handling</td>
</tr>
<tr>
<td>—logical</td>
</tr>
<tr>
<td>—I/O</td>
</tr>
<tr>
<td>• Comments</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TABLE II—Sample Metrics*</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logical Complexity</td>
</tr>
<tr>
<td>$M_{\text{LOGC}} = \frac{LS}{TS}$</td>
</tr>
<tr>
<td>where: $LS = \text{Number of logical (branch creating) statements}$ $TS = \text{Total statements in routine}$</td>
</tr>
<tr>
<td>Logical Statement Complexity</td>
</tr>
<tr>
<td>$M_{\text{LOGS}} = \frac{TB}{LS}$</td>
</tr>
<tr>
<td>where: $TB = \text{Total branches in the routine}$ $LS = \text{Number of logical statements}$</td>
</tr>
<tr>
<td>Loop Nesting Complexity**</td>
</tr>
<tr>
<td>$M_{\text{LOOP}} = \sum m_i W_i$</td>
</tr>
<tr>
<td>where: $i = \text{nesting level (1st is non-nested)}$ $m = \text{number of loops}$</td>
</tr>
<tr>
<td>$W_i = \sum_{i=1}^{m} \sum_{j=1}^{n} 4^{0(i-j)}$</td>
</tr>
</tbody>
</table>

Attributes and metrics

Attributes identify specific software characteristics. These might involve such things as difficulty, complexity, and readability. Metrics quantify attributes. They can range from simple counts of source statements, logic branches, and routine interfaces to calculated values of difficulty, complexity, test sufficiency, and readability, these latter calculated metrics requiring mathematical expressions of definition. Metrics are being used in understanding and explaining error histories, and it is hoped that they will eventually provide means for comparing varying and unrelated software projects. Table II presents several examples of metrics considered in the Software Reliability Study.

Of course, the definition of such things as complexity is judgmental on the part of the analyst and not altogether independent of what data are available. Some metrics have shown nothing when an attempt is made to correlate them with error frequencies; others can't be supported with real data for lack of collection tools. We are forced to admit that evaluation of metrics is part of the study process, and it will take several iterations on a number of differing software systems to identify the more universally applicable metrics. One point that should be made, however, is that every attempt to define metrics, even if unsuccessful, adds to the analyst's understanding of the software under study. Every attempt to understand what makes software complex, more readable, easier to test, or easier to maintain aids in the understanding of software errors and provides a potential for uncovering improved development or test techniques.

RESULTS FROM SOFTWARE RELIABILITY ANALYSIS

Four examples are given below to indicate ways in which understanding of software and the development process can be improved through analysis of empirical reliability data.
Detecting on-going problems

During the formal test period for one of the projects examined in the Software Reliability Study, it was discovered that a portion of the software system was judged to be very error prone. Using a simple count of errors discovered as a gauge this software looked no worse than other software in the system. However, by also considering the criticality and type of the relatively few errors being encountered, the system engineering group would have been able to identify a problem traceable to rapidly changing requirements and the need for substantial redesign and test prior to delivery. The symptoms of the problem were present throughout the development cycle (even as early as preliminary design), but there was no effort to create an historical dossier of all available data and tie it to the offending software. As a result, the severity of the problem was not realized and the software continued to be a problem even after delivery.

Benefits of new software development techniques

One of the projects analyzed in the Software Reliability Study utilized a number of advanced development and test techniques and tools. Among these were structured programming, a 100 executable statement limit on routine size, programming standards enforced by audit tools, and rigorous development testing requirements, including a requirement to execute all code at the routine level. Test tools include a dynamic test monitor to track the amount of code exercised. Preliminary results* of a comparison of this and other projects show a tendency for errors to be found sooner and for the total error count to be lower. Figure 2 depicts an actual cumulative error history (solid curve) of a project using only standard development techniques and the suspected results of using new development and test techniques (dashed curve).

Other preliminary results of error analysis show a lower percentage of the total data processing errors* using newer techniques, presumably because of the smaller, better structured, and easier to test routines. On the other hand, the percentage of interface errors is greater for the system with many smaller routines.

Understanding error rates

Hopefully, a few (ideally one) independent variables correlate well with the number of errors in the software and enable us to use the independent variables values to understand ultimate reliability. Our approach has been to use a large sample of real instances to compute the correlation between metrics characterizing the software and the number of errors in the software. The metric that correlated best with the occurrence of actual software errors was the simple measure of size in source statements. (Routine complexity expressed as a count of logic branches showed similar trends.)

Although correlation for all routines in the software system taken together was poor (correlation coefficient = 0.304), selected groupings of routines within the system, specifically routines which function together to form parts of each software subsystem, correlated very well. The highest correlation coefficient encountered was 0.910 for a straight line curve fit of 25 routines ranging in size from 10 to 2300 source statements, but the bulk of the correlation coefficients were on the order of 0.7. Although slopes varied from group to group and the best fit was not always linear, a typical data point was 20 errors/1000 source statements.

Results from other metric analyses, aside from providing insight on what makes software complex and how to measure this complexity, tend to show that such things as schedule pressures, the availability of needed resources, and requirements have as much or more to do with the eventual quality of the software as do the structural characteristics of the system. For instance, routines which exhibited similar error histories but were quite different in structure and software type were subject to the same external influences (e.g., they were produced by the same developers, suffered from poorly defined or rapidly changing requirements, lacked realistic data base values for testing, etc.). Unfortunately, these influences are not easily quantifiable through present techniques, but their effects are seen in the data.

Error sources

One of the most enlightening results of the Software Reliability Study has centered on where errors were introduced. Although all errors detected past the design phase are found in the code during some form of checkout or testing, these errors need not originate from the coding phase. That is, they may be due to design or even requirements definition errors. For one of the systems analyzed in the Software Reliability Study (a command and control system) accurate

* Results are considered preliminary because, at the time of writing, the project was still under development and the total error history was not complete. We eagerly await completion of this project when our hypothesis testing can be completed.

* For example, iterative procedure, bit manipulation, and indexing errors.
TABLE III—Relative Frequency of Design and Coding Errors

<table>
<thead>
<tr>
<th>Modification</th>
<th>No. of Source Statements in Modification</th>
<th>Total Errors Encountered</th>
<th>% Design Errors</th>
<th>% Coding Errors</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1253</td>
<td>152</td>
<td>73.6</td>
<td>26.4</td>
</tr>
<tr>
<td>B</td>
<td>9880</td>
<td>156</td>
<td>73.7</td>
<td>26.3</td>
</tr>
<tr>
<td>C</td>
<td>779</td>
<td>73</td>
<td>35.6</td>
<td>64.4</td>
</tr>
<tr>
<td>D</td>
<td>9631</td>
<td>419</td>
<td>51.6</td>
<td>48.4</td>
</tr>
<tr>
<td>E</td>
<td>4575</td>
<td>199</td>
<td>58.8</td>
<td>41.2</td>
</tr>
<tr>
<td>F</td>
<td>--</td>
<td>113</td>
<td>61.9</td>
<td>38.1</td>
</tr>
<tr>
<td>G</td>
<td>--</td>
<td>120</td>
<td>65.8</td>
<td>34.2</td>
</tr>
</tbody>
</table>

*Size data not accurate.

Data was available which identified a failure as due to either a design or coding error. No data were available to identify those errors stemming from poor requirements, but these errors were considered to be negligible. Data accumulated over a three year period in seven modifications to the system is summarized in Table III.

This preponderance of design errors is supported by data from the three other systems also, although percentage breakdowns are not so easily obtainable due to inaccuracies in the data collection process. Among the more common errors were those where the design did not take into account certain aspects of the physical problem to be solved. Software failures, in this case, manifested themselves in a number of ways; the error was in the design in the form of missing logic or condition tests. In one system analyzed, where approximately 40 percent of the budget was spent on design, 9.7 percent of a total of 4439 reported errors were of the type described above (i.e., the approved design was missing some necessary logic). Two other systems exhibited 6.8 percent (out of a total of 1498) and 13.4 percent (out of a total of 539) for the same type of error. Typically it is this type of error, a fundamental design error, that is most difficult (and costly) to fix. The cost involved in diagnosing and correcting design and coding errors has been quantified by Shooman and Bolsky. In their paper the effort in manhours to diagnose and correct design and coding errors were compared. Results showed that design errors required an average of 3.1 manhours to diagnose and correct design and coding errors were compared. Results showed that design errors required an average of 3.1 manhours to diagnose and correct design and coding errors were compared. Results showed that design errors required an average of 3.1 manhours to diagnose and correct design and coding errors were compared. Results showed that design errors required an average of 3.1 manhours to diagnose and correct design and coding errors were compared. Results showed that design errors required an average of 3.1 manhours to diagnose and correct design and coding errors were compared. Results showed that design errors required an average of 3.1 manhours to diagnose and correct design and coding errors were compared. Results showed that design errors required an average of 3.1 manhours to diagnose and correct design and coding errors were compared. Results showed that design errors required an average of 3.1 manhours to diagnose and correct design and coding errors were compared. Results showed that design errors required an average of 3.1 manhours to diagnose and correct design and coding errors were compared. Results showed that design errors required an average of 3.1 manhours to diagnose and correct design and coding errors were compared. Results showed that design errors required an average of 3.1 manhours to diagnose and correct design and coding errors were compared. Results showed that design errors required an average of 3.1 manhours to diagnose and correct design and coding errors were compared. Results showed that design errors required an average of 3.1 manhours to diagnose and correct design and coding errors were compared. Results showed that design errors required an average of 3.1 manhours to design and coding errors were compared. Results showed that design errors required an average of 3.1 manhours to design and coding errors were compared. Results showed that design errors required an average of 3.1 manhours to design and coding errors were compared. Results showed that design errors required an average of 3.1 manhours to design and coding errors were compared. Results showed that design errors required an average of 3.1 manhours to design and coding errors were compared. Results showed that design errors required an average of 3.1 manhours to design and coding errors were compared. Results showed that design errors required an average of 3.1 manhours to design and coding errors were compared. Results showed that design errors required an average of 3.1 manhours to design and coding errors were compared. Results showed that design errors required an average of 3.1 manhours to design and coding errors were compared. Results showed that design errors required an average of 3.1 manhours to design and coding errors were compared. Results showed that design errors required an average of 3.1 manhours to design and coding errors were compared. Results showed that design errors required an average of 3.1 manhours to design and coding errors were compared. Results showed that design errors required an average of 3.1 manhours to design and coding errors were compared. Results showed that design errors required an average of 3.1 manhours to design and coding errors were compared. Results showed that design errors required an average of 3.1 manhours to design and coding errors were compared. Results showed that design errors required an average of 3.1 manhours to design and coding errors were compared. Results showed that design errors required an average of 3.1 manhours to design and coding errors were compared. Results showed that design errors required an average of 3.1 manhours to design and coding errors were compared. Results showed that design errors required an average of 3.1 manhours to design and coding errors were compared. Results showed that design errors required an average of 3.1 manhours to design and coding errors were compared. Results showed that design errors required an average of 3.1 manhours to design and coding errors were compared. Results showed that design errors required an average of 3.1 manhours to design and coding errors were compared. Results showed that design errors required an average of 3.1 manhours to design and coding errors were compared. Results showed that design errors required an average of 3.1 manhours to design and coding errors were compared. Results showed that design errors required an average of 3.1 manhours to design and coding errors were compared.

With design errors being more costly and more frequent than coding errors, the conclusion is that improvements in design technique and design aids or tools are needed. The coder has the compiler to help him, and the tester, in recent years, has had automated test tools. The designer presently has only his wits to help him.

NEEDED DATA

Key to the empirical approach of determining software reliability is the existence of good (well organized and defined) data. Presently data collection schemes don’t provide enough of the right kind of data, nor are the data collected at the right time.

Figure 3 illustrates a typical software development project by phase and points out when various types of data can become available. Ovals indicate error or problem data that can be produced and collected. Ancillary data, shown below the phases of the development cycle, become available roughly at the times indicated, are absolutely necessary to the understanding of the error histories, and can be extremely useful in assessing on-going project performance. Triangles along the base of Figure 3 denote points at which snapshots of the software structural characteristics can be taken to gain a picture of the volume of change.

Individual data items will vary from project to project, and most projects have a potential for creating a tremendous amount of data. From our work in collecting and analyzing these data, three general categories appear to be most important in answering both reliability and status related questions. These, along with needed data within each category, are given below.

Error data

Information concerning errors is generated throughout the development cycle of the project. No matter which phase in the cycle provides this information, data should be recorded on collection forms tailored to the data and should be collected as near the identification date as possible. Furthermore, categorization of errors by type should be performed by the problem fixer, if at all possible. Questions that should be addressed in the collection process are the following:

- When was the error introduced? Was the error made during requirements definition, software design, coding, testing or operational maintenance?
- How critical to successful operation of the software system is the error? This amounts to a priority on the need

Figure 3—Software development and reliability data

From the collection of the Computer History Museum (www.computerhistory.org)
for corrective action and is useful as a normalizing factor in comparing data from different projects.

- How and when was the error found? This is useful in evaluating the software development and test processes.
- What test “stress” was applied to produce the failure? This is important to answering the question “How much testing was done?”
- Was the error independent of other errors or the result of a previous fix? This information is important in assessing the quality of software maintenance activities.
- What was the resource expenditure in manhours and machine time to diagnose and correct an error?

Characteristics of testing

It is testing that produces the bulk of the software problem reports which serve as the basis for analysis. The number of problems documented will depend to a great degree on the formality of testing and when this formality is initiated. Regardless of when this control is imposed, it is important to be able to relate software failure data to specific dynamic executions of specific test procedures. If these test procedures are then evaluated according to the amount of stress they place on the software, an attempt can be made to determine when enough testing has been done and when the quality of the software is resulting from a particular test program. Measures of stress vary from CPU time per test execution to the number of program segments exercised and the representativeness of the input data base, and the worth of any of these measures depends upon who you talk to. The important point here is to know how much testing, according to some measure, produced a particular error history. Test stress has been and remains to be difficult data to collect. Both the tools and the methodology for collection are in their infancy, but recent work in software reliability modeling offers promise in this area.

Characteristics of the development cycle

These characteristics will vary from project to project but can generally be grouped according to schedules, resources, and personnel. Although important to the understanding of error trends and reliability related data, it is the characteristics of the development cycle itself that often point to problems that are documented on error reports later in the project. Project resource availability can be a powerful tool in identifying problems. For example, unavailability of computer time during the compilation and debug phase may, to meet schedules, force reduction of the amount of detailed development testing a critical routine undergoes prior to formal testing or delivery to a customer. Early identification of this situation could allow additional preliminary testing steps to be taken prior to beginning of the more formal testing involving that routine. Parameters useful in characterizing the development cycle are given in Table IV below.

CONCLUSION

Data collection and analysis represent an added workload for the project providing the data; however, the yield of useful information for both technical and management control will be increased. Done properly, manpower would be provided for collection of data throughout the life of the project; much of the useful data is perishable and must be collected as it is created. Further project involvement is required in the analysis of the data because individual performers alone are able to provide some of the data with sufficient accuracy (e.g., causative error data). Access to project experience is essential to accurate interpretation, especially if the analysis is done independently by other than project performers.

Collecting and analyzing data makes it possible to answer questions concerning software reliability with something other than philosophy and speculation. Increased awareness of the types of errors encountered, the characteristics of the development process that produced them, and the conditions under which they occurred is the first step in quantitatively specifying measures of quality to be used by purchasers and producers of software alike.

BIBLIOGRAPHY


