SOFTWARE METRICS AND RELIABILITY

Dr. Linda Rosenberg	Ted Hammer	Jack Shaw
Unisys/NASA GSFC	NASA GSFC	NASA GSFC
Bld 6 Code 300.1	Bld 6 Code 302	Bld 6 Code 302
Greenbelt, MD 20771	Greenbelt, MD 20771	Greenbelt, MD 20771
USA	USA	USA
301-286-0087	301-286-7475	301-286-7123
Linda.Rosenberg@gsfc.nasa.gov	thammer@pop300.gsfc.nasa.gov	jjshaw@pop300.gsfc.nasa.gov

Abstract

The IEEE defines reliability as "The ability of a system or component to perform its required functions under stated conditions for a specified period of time." To most project and software development managers, reliability is equated to correctness, that is, they look to testing and the number of "bugs" found and fixed. While finding and fixing bugs discovered in testing is necessary to assure reliability, a better way is to develop a robust, high quality product through all of the stages of the software lifecycle. That is, the reliability of the delivered code is related to the quality of all of the processes and products of software development; the requirements documentation, the code, test plans, and testing.

Software reliability is not as well defined as hardware reliability, but the Software Assurance Technology Center (SATC) at NASA is striving to identify and apply metrics to software products that promote and assess reliability. This paper discusses how NASA projects, in conjunction with the SATC, are applying software metrics to improve the quality and reliability of software products. Reliability is a by-product of quality, and software quality can be measured. We will demonstrate how these quality metrics assist in the evaluation of software reliability. We conclude with a brief discussion of the metrics being applied by the SATC to evaluate the reliability .

I. Definitions

Software cannot be seen nor touched, but it is essential to the successful use of computers. It is necessary that the reliability of software should be measured and evaluated, as it is in hardware. IEEE 610.12-1990 defines reliability as "The ability of a system or component to perform its required functions under stated conditions for a specified period of time." IEEE 982.1-1988 defines Software Reliability Management as "The process of optimizing the reliability of software through a program that emphasizes software error prevention, fault detection and removal, and the use of measurements to maximize reliability in light of project constraints such as resources, schedule and performance." Using these definitions, software reliability is comprised of three activities:

1. Error prevention
2. Fault detection and removal
3. Measurements to maximize reliability, specifically measures that support the first two activities

There has been extensive work in measuring reliability using mean time between failure and mean time to failure.[1] Successful modeling has been done to predict error rates and

reliability.[1,2,3] These activities address the first and third aspects of reliability, identifying and removing faults so that the software works as expected with the specified reliability. These measurements have been successfully applied to software as well as hardware. But in this paper, we would like to take a different approach to software reliability, one that addresses the second aspect of reliability, error prevention.

II. Errors, Faults and Failures

The terms errors, faults and failures are often used interchangeable, but do have different meanings. In software, an error is usually a programmer action or omission that results in a fault. A fault is a software defect that causes a failure, and a failure is the unacceptable departure of a program operation from program requirements. When measuring reliability, we are usually measuring only defects found and defects fixed.[4] If the objective is to fully measure reliability we need to address prevention as well as investigate the development starting in the requirements phase – what the programs are developed to.

It is important to recognize that there is a difference between hardware failure rate and software failure rate. For hardware, as shown in Figure 1, when the component is first manufactured, the initial number of faults is high but then decreases as the faulty components are identified and removed or the components stabilize. The component then enters the useful life phase, where few, if any faults are found. As the component physically wears out, the fault rate starts to increase.[1]

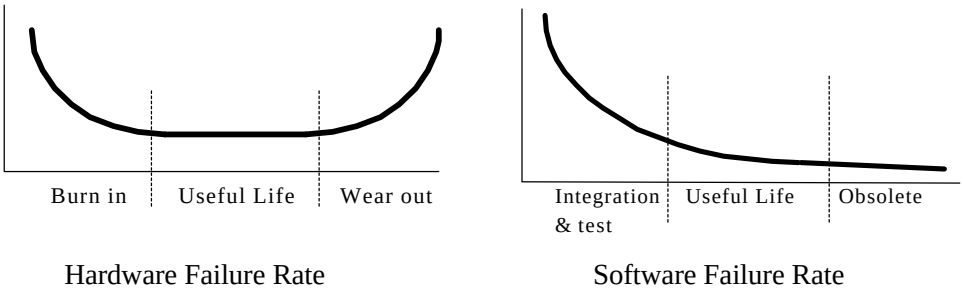

Figure 1: Failure Rates

Software however, has a different fault or error identification rate. For software, the error rate is at the highest level at integration and test. As it is tested, errors are identified and removed. This removal continues at a slower rate during its operational use; the number of errors continually decreasing, assuming no new errors are introduced. Software does not have moving parts and does not physically wear out as hardware, but is does outlive its usefulness and becomes obsolete.[5]

To increase the reliability by preventing software errors, the focus must be on comprehensive requirements and a comprehensive testing plan, ensuring all requirements are tested. Focus also must be on the maintainability of the software since there will be a "useful life" phase where sustaining engineering will be needed. Therefore, to prevent software errors, we must:

1. Start with the requirements, ensuring the product developed is the one specified, that all requirements clearly and accurately specify the final product functionality.
2. Ensure the code can easily support sustaining engineering without infusing additional errors.

3. A comprehensive test program that verifies all functionality stated in the requirements is included.

Reliability as a Quality Attribute

There are many different models for software quality, but in almost all models, reliability is one of the criteria, attribute or characteristic that is incorporated. ISO 9126 [1991] defines six quality characteristics, one of which is reliability. IEEE Std 982.2-1988 states "A software reliability management program requires the establishment of a balanced set of user quality objectives, and identification of intermediate quality objectives that will assist in achieving the user quality objectives." [6] Since reliability is an attribute of quality, it can be concluded that software reliability depends on high quality software.

Building high reliability software depends on the application of quality attributes at each phase of the development life cycle with the emphasis on error prevention, especially in the early life cycle phases. Metrics are needed at each development phase to measure applicable quality attributes. IEEE Std 982.2-1988 includes the diagram in Figure 2, indicating the relationship of reliability to the different life cycle phases.[7]

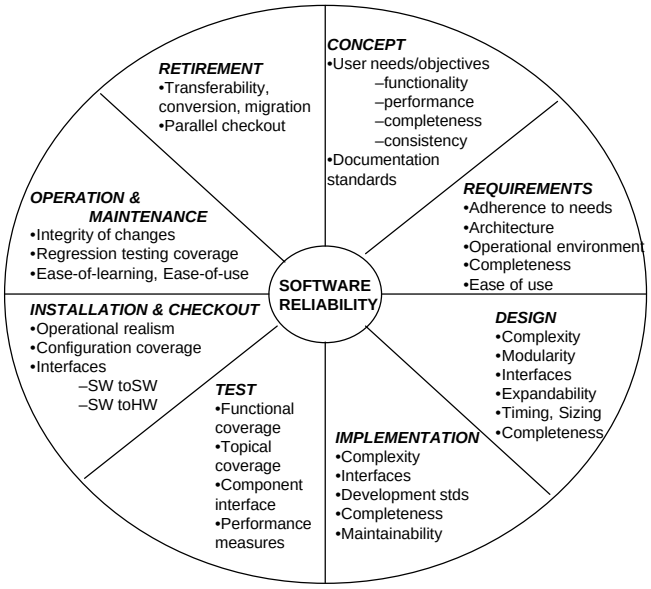

Figure 2: Quality Factors Impacting Reliability

In focusing on error prevention for reliability, we need to identify and measure the quality attributes applicable at different life cycle phases. As discussed previously, we need to specifically focus on requirements, design, implementation, and test phases.

IV. Software Metrics for Reliability

Software metrics are being used by the Software Assurance Technology Center (SATC) at NASA to help improve the reliability by identifying areas of the software requirements specification and code that can potentially cause errors. The SATC also examines the test plan for complete requirement coverage without excessive (and expensive) testing. The remainder of this paper discusses metrics used by the SATC for identifying potential errors before software is

released. NASA project data is used to demonstrate that application of the metrics. We will address three life cycle phases where error prevention techniques and software metrics can be applied to impact the reliability: requirements, coding, and testing.

A. Requirements Reliability Metrics

Requirements specify the functionality that must be included in the final software. It is critical that the requirements be written such that is no misunderstanding between the developer and the client. Using the quality attributes for reliability shown in Figure 2, for high reliability software, the requirements must be structured, complete, and easy to apply.

There are three primary formats for requirement specification structure, by IEEE, DOD and NASA.[7,8,9] These specify the content of the requirement specification outside the requirements themselves. Consistent use of a format such as these ensures that critical information, such as operational environment, is not omitted.

Complete requirements are stable and thorough, specified in adequate detail to allow design and implementation to proceed. Requirement specifications should not contain phrases such at TBD (to be determined) or TBA (to be added) since the lack of specificity of these phrases may have a negative impact on the design, causing a disjointed architecture.

To increase the ease of using requirements, they are usually written in English [vice a specialized writing style (e.g., Z notation)], which can easily produce ambiguous terminology. In order to develop reliable software from the requirements phase forward, the requirements must not contain ambiguous terms, or contain any terminology that could be construed as an optional requirement. Ambiguous requirements are those that may have multiple meanings or those that seem to leave to the developer the decision whether or not to implement a requirement.

The importance of correctly documenting requirements has caused the software industry to produce a significant number of aids to the creation and management of the requirements specification documents and individual specification statements. However very few of these aids assist in evaluating the quality of the requirements document or the individual specification statements themselves. The SATC has developed a tool to parse requirement documents. The Automated Requirements Measurement (ARM)[1] software was developed for scanning a file that contains the text of the requirement specification. During this scan process, it searches each line of text for specific words and phrases. These search arguments (specific words and phrases) are indicated by the SATC's studies to be an indicator of the document's quality as a specification of requirements. ARM has been applied to 56 NASA requirement documents. Seven measures were developed, as shown below.

1. Lines of Text - Physical lines of text as a measure of size.
2. Imperatives - Words and phases that command that something must be done or provided. The number of imperatives is used as a base requirements count.
3. Continuances -Phrases that follow an imperative and introduce the specification of requirements at a lower level, for a supplemental requirement count.
4. Directives – References provided to figures, tables, or notes.

[1] ARM is available free from the SATC homepage: http://satc.gsfc.nasa.gov

5. Weak Phrases - Clauses that are apt to cause uncertainty and leave room for multiple interpretation measure of ambiguity.
6. Incomplete – Statements within the document that have TBD (To be Determined) or TBS (To Be Supplied).
7. Options - Words that seem to give the developer latitude in satisfying the specifications but can be ambiguous.

It must be emphasized that the tool does not attempt to assess the correctness of the requirements specified. It assesses individual specification statements and the vocabulary used to state the requirements; it also has the capability to assess the structure of the requirement document. [10]

The results of the analysis of the Design Level 3 and Detailed Level 4 requirements are shown in Table 1 with the comparison to other NASA documents.

56 DOCUMENT	**LINES OF TEXT** - Count of the physical lines of text	**Imperatives** - shall, must, will, should, is required to, are applicable, responsible for	**Continuances** - as follows, following, listed, inparticular, support	**Directives** - figure, table, for example, note:	**Weak Phrases** - adequate, as applicable, as appropriate, as a minimum, be able to, be capable, easy, effective, not	**Incomplete** - TBD, TBS, TBR	**Options** - can, may, optionally
Average	4772	682	423	49	70	25	63
Level 3	1011	588	577	10	242	1	5
Level 4	1432	917	289	9	393	2	2

Table 1 : Textual Requirement Analysis

We are especially concerned with the number of weak phrases since the contract is bid using Design Level 3 and acceptance testing will be against these requirements. It is also of concern that the number of weak phrases has increased in Detailed Level 4, the requirements used to write the design and code and used in Integration testing.

ARM also evaluates the structure of the document by identifying the number of requirements at each level of the hierarchical numbering structure. This information may serve as an indicator of a potential lack of structure.[10] Lack of structure impacts reliability by increasing the difficulty in making changes and the possible inappropriate level of detail may artificially constrain portions of design.

B. Design and Code Reliability Metrics

Although there are design languages and formats, these do not lend themselves to an automated evaluation and metrics collection. The SATC analyzes the code for the structure and architecture to identify possible error prone modules based on complexity, size, and modularity.

It is generally accepted that more complex modules are more difficult to understand and have a higher probability of defects than less complex modules.[5] Thus complexity has a direct impact

on overall quality and specifically on maintainability. While there are many different types of complexity measurements, the one used by the SATC is logical (Cyclomatic) complexity, which is computed as the number of linearly independent test paths.

Size is one of the oldest and most common forms of software measurement. Size of modules is itself a quality indicator. Size can be measured by: total lines of code, counting all lines; non-comment non-blank which decreases total lines by the number of blanks and comments; and executable statements as defined by a language dependent delimiter.

The SATC has found the most effective evaluation is a combination of size and complexity. The modules with both a high complexity and a large size tend to have the lowest reliability. Modules with low size and high complexity are also a reliability risk because they tend to be very terse code, which is difficult to change or modify.[11]

Although these metrics are also applicable to object oriented code, additional metrics are needed to evaluate the quality of the object oriented structure. The SATC uses the following metrics for object oriented quality analysis: Weighted methods per class (WMC), Response for a Class (RFC), Coupling Between Objects (CBO), Depth in Tree (DIT), and Number of Children (NOC). Since there are very few industry guidelines for these metrics, the SATC has developed guidelines based on NASA data. An example is Weighted Methods per Class.

Weighted Methods per Class is a predictor of how much time and effort is required to develop and maintain the class. The higher the WMC, the more testing necessary and maintenance is increased. In general, classes should have less than 20 methods, but up to 40 is acceptable. The complexity of a method should not exceed 5. Therefore, WMC is preferred less than 100 and should not exceed 200. Classes with higher WMC will require extensive testing for comprehensive coverage. They also will be difficult to maintain. Although further examination and metrics are needed, we may conclude with additional information that these classes have a low reliability .[12]

C. Testing Reliability Metrics

Testing metrics must take two approaches to comprehensively evaluate the reliability. The first approach is the evaluation of the test plan, ensuring that the system contains the functionality specified in the requirements. This activity should reduce the number of errors due to lack of expected functionality. The second approach, one commonly associated with reliability, is the evaluation of the number of errors in the code and rate of finding/fixing them. The SATC has developed a model to simulate the finding of errors and projects the number of remaining errors and when they will all be identified.

To ensure that the system contains the functionality specified, test plans are written that contain multiple test cases; each test case is based on one system state and tests some functions that are based on a related set of requirements. The objective of an effective verification program is to ensure that every requirement is tested, the implication being that if the system passes the test, the requirement's functionality in included in the delivered system. An assessment of the traceability of the requirements to test cases is needed.

In the total set of test cases, each requirement must be tested at least once, and some requirements will be tested several times because they are involved in multiple system states in varying scenarios and in different ways. But as always, time and funding are issues; while each requirement must be comprehensively tested, limited time and limited budget are always constraints upon writing and running test cases. It is important to ensure that each requirement is adequately, but not excessively, tested. In some cases, the requirements can be grouped together using criticality to mission success as their common thread; these *must* be extensively tested. In other cases, requirements can be identified as low criticality; if a problem occurs, their functionality does not affect mission success while still achieving successful testing.[13]

From an error trending perspective, the model developed by the SATC[2] employs a straightforward implementation of the Musa Model to compute a nonlinear approximation to the cumulative errors found to date. Rather than using the probabilistic approach to determine model parameters, however, we apply an "iterative" process that employs a non-linear optimization technique to fit the model-generated cumulative error distribution curve to the actual cumulative error distribution curve. A sum-of-difference-squared is used to determine the "best fit." Using the computed values and other functions defined by Musa allow us to estimate: (a) the number of errors remaining in the product, and (b) the time needed to detect the remaining errors.[1]

To relate manpower utilization to detecting the errors remaining in the product, we use a modified version of the Musa Model where the linear term in the exponential is replaced by the integral of a Rayleigh function. Manpower utilization and cumulative manpower utilization curves can be computed, and then extrapolated to include the manpower needed to detect the remaining errors. We will not discuss this work to any extent in this paper since it is uses many of the accepted reliability approaches and the tool is still under development.

V. Conclusion

Metrics to measure software reliability do exist and can be used starting in the requirements phase. At each phase of the development life cycle, metrics can identify potential areas of problems that may lead to problems or errors. Finding these areas in the phase they are developed decreases the cost and prevents potential ripple effects from the changes, later in the development life cycle. Metrics used early can aid in detection and correction of requirement faults that will lead to prevention of errors later in the life cycle. The cost benefits of finding and correcting problems in the requirements phase has been demonstrated to be at least a factor of 14, making a strong argument for pursuing this approach and building in reliability starting at the requirements phase.

REFERENCES

[1] Musa, J.D., A. Iannino and K. Okumoto, *Software Reliability: Measurement, Prediction, Application*, Professional Edition: Software Engineering Series, McGraw–Hill, New York, NY., 1990.

[2] This model was developed in conjunction with Sean Arthur from Virginia Tech and Darush Davani from Towson University.

[2] Triantafyllos, G., S. Vassiliadis and W. Kobrosly, “On the Prediction of Computer Implementation Faults Via Static Error Prediction Models,” Journal of Systems and Software, Vol. 28, No. 2, February 1995, pp. 129-142.

[3] Waterman, R.E and L.E. Hyatt, “Testing - When Do I Stop?”, (Invited) International Testing and Evaluation Conference, Washington, DC, October 1994.

[4] IEEE Standard 610.12-1990 Glossary of Software Engineering Terminology

[5] Kitchenham, Barbara, Pfleeger, Shari Lawrence, Software Quality: The Elusive Target, *IEEE Software 13, 1* (January 1996) 12-21.

[6] Gillies, A.C., Software Quality, Theory and management, Chapman Hall Computing Series, London, UK, 1992.

[7] IEEE Standard 982.2-1987 Guide for the Use of Standard Dictionary of Measures to Produce Reliable Software.

[8] NASA Software Assurance guidebook, NASA GSFC MD, Office of Safety and Mission Assurance, 1989.

[9] Department of Defense. *Military Standard Software Development and Documentation* (December 1994), MIL-STD-498.

[10] Wilson, W., Rosenberg, L., Hyatt, L., Automated Quality Analysis of Natural Language Requirement Specifications in *Proc. Fourteenth Annual Pacific Northwest Software Quality Conference*, Portland OR, 1996.

[11] Rosenberg, L., and Hammer, T., Metrics for Quality Assurance and Risk Assessment, Proc. Eleventh International Software Quality Week, San Francisco, CA, 1998.

[12] Rosenberg, L., Metrics for Object Oriented Environment, EFAITP/AIE Third Annual Software Metrics Conference, 1997.

[13] Hammer, T., Rosenberg, L., Huffman, L., Hyatt, L., Measuring Requirements Testing in *Proc. International Conference on Software Engineering* (Boston MA, May 1997) IEEE Computer Society Press.

[14] Boehm, B. Tutorial: Software Risk Management (1989), IEEE Computer Society Press.