Measuring the Effect of Design Decisions on Software Reliability

Jeffrey Stineburg	Wayne Zage	Dolores Zage
Senior Software Engineer	Director, SERC	Res. Coordinator, SERC
Software Quality Assurance	CS Department	CS Department
Raytheon Corporation	Ball State University	Ball State University
Fort Wayne, IN	Muncie, IN 47306	Muncie, IN 47306
1-260-429-6365	1-765-285-8664	1-765-285-8646
Jeffery_Stineburg@raytheon.com	wmzage@bsu.edu	dmzage@bsu.edu

Abstract

This paper presents a model for estimating the effect of design decisions on software reliability based on design metrics developed in the Software Engineering Research Center (SERC). The paper introduces the concepts of design significance and stress points, and a method to identify and measure these in software. After a brief overview of selected software reliability models, the problem of validating life-critical software is presented. The paper then investigates the proposition that a relationship exists between the design metric D(G) and the defects that are found in the field. A study performed on a subset of a large defense software system provides empirical evidence to support the proposition. The last section of the paper describes a high reliability engineering process that has been developed based on the concepts in this paper. The process is implemented on an active defense software development program.

Key Words: software reliability, design metrics, high reliability engineering process, software stress, design significance.

1. Defining Software Reliability

In the history of the software industry, the point at which the reliability of a software program first became an issue is not known, although it seems reasonable to assume that the issue arose shortly after the first program was written. Butler and Finelli [1] provide the generally accepted definitions for reliability categories given in Table 1, Column 2. Ultra-reliable software applications include all safety-critical or life-critical systems. In particular, this pertains to defense, aviation, space, medical, and life-support systems.

Table 1. Reliability Categories

CATEGORY	FAILURE RATE (PER HOUR)	DEFECT RATE (DEFECTS/KSLOC)
Ultra-reliability	$< 10^{-7}$	0.1
Moderate Reliability	10^{-3} to 10^{-7}	0.5
Low Reliability	$> 10^{-3}$	1

It is also useful to tie defect rates to the reliability categories in Table 1. Defect rates that were determined to be reasonable during the NASA and IBM Federal Systems programs for such applications as the space station, the space shuttle and large commercial systems are given in Table 1, Column 3 [5]. Since our research focuses on designing software to produce an expected reliability, we will focus on reliability as related to defect rates in this paper. The software reliability expression used in this paper is derived from Dunn [6]:

R = 1- (number_of_defects (predicted or actual) / lines_of_code)

Thus, the reliability of a 300,000 line software program containing (or predicted to contain) 5 defects per KSLOC (1500 defects total) is:

R = 1 – (5*300/300,000) = 1 – (1500/300,000) = 1 - .005 = .995

This program would fall into the low reliability category of Table 1.

2. Software Reliability Models

Many software reliability models have been developed to predict the reliability of a software product. Nearly all are based either on failure rate or time between failures, and most use measures obtained during testing. In some cases the models attempt to predict a latent defect count using measures obtained during earlier phases, which is then translated into a reliability figure. We survey some of the models here, showing the diversity of reliability modeling.

2.1 Complexity Models

Models based on program complexity have been developed to predict software defects. The most well known of these are McCabe's cyclomatic complexity metric V(G), and Halstead's Volume metric V. Lennselius [11] analyzed a number of complexity metrics collected for a telecommunications system and computed the correlation coefficients among them. The results are shown in Table 2. The metrics shown in the table are as follows: the number of SDL symbols used, SDL; the cyclomatic complexity metric, V(G); the maximum nesting level, NC; a count of the total program branches, B; Halstead's volume metric, V; and total lines of code, LOC.

Table 2. Correlation Coefficients for Lennselius Data

METRIC	SDL	V(G)	NC	B	V	LOC
SDL						
V(G)	.99					
NC	.89	.92				
B	.98	.99	.95			
V	.91	.92	.89	.92		
LOC	.93	.94	.93	.94	.96	
DEFECTS	.88	.90	.85	.89	.94	.87

Note that all the correlation coefficients are above 0.84. This reflects a high degree of multicollinearity among the metrics, and is a strong indication that all are based on a common underlying factor - LOC. Analysis indicates that any one of these metrics can be used equally well in a regression model for the prediction of defects.

2.2 Exponential Models

Weibull exponential model has its basis in hardware reliability and was one of the earliest applied to software [8]. A member of this family, the Rayleigh model, is gaining popularity for use during development and is incorporated in a number of tools (e.g., SWEEP, a software error estimation program from SPC).

2.3 Test Coverage Models

Many studies estimate software reliability from the standpoint of testing. This is a different approach from the conventional software reliability growth models. In many experiments a positive linear relation was uncovered independent of program size, fault distribution and the coverage criterion. There are several techniques that incorporate test coverage measurement into the estimation of software reliability, such as those by Chen, Lyu and Wong [2, 3]. Chen, Wong and Pasquini further extend the coverage technique by first estimating testability and evaluating how well the software was written into the process [23]. This method was shown to possess better predictive ability when compared to conventional growth models. A variation of test coverage reliability modeling has been studied by Mathur and Krishnamurthy, where they have estimated the reliability of a software system by estimating the reliabilities of its components [14].

2.4 Bayesian Estimation Models

Bayesian estimation models are based on the concept of fault-free operation and incorporate the data from previous intervals as well as the current interval. They are gaining in popularity and are the subject of many research projects. One of the problems with these models is the subjectivity that can exist in the prior assumptions, although with the growing base of accurate historical data in CMM/CMMI rated companies, the subjectivity problem is diminishing.

2.5 Process Quality Models

With the increasing adoption of the Capability Maturity Model (CMM) in the software development industry there has been a corresponding interest in relating the CMM level to the latent defects at delivery of the software. A similar interest is being shown in relating the use of the ‘Cleanroom’ method to defect density [7]. Diaz and Sligo [4] have been able to show empirical evidence supporting the CMM-level/Defect-density relationship. A number of other studies have refined the relationship, as shown in Table 3 [20]. The last column in Table 3 gives the defect plateau level for each of the CMM levels of maturity. The defect plateau level is the level at which the number of defects removed is balanced by an equivalent number of defects inserted by the correction process itself. Industry data indicates that product stabilization occurs at about 48 months after delivery for initial product releases [24]. Subsequent releases of the same product tend to stabilize at about 24 months after delivery.

Table 3. CMM Level and Defect Density at Delivery

CMM LEVEL	FAULTS / KSLOC (Keene)	FAULTS / KSLOC (Jones)	FAULTS / KSLOC (Krasner)	Defect Plateau Level 48 months after initial delivery or 24 months following subsequent deliveries
V	0.5	0.5	0.5	1.5%
IV	1.0	1.4	2.5	3.0%
III	2.0	2.69	3.5	5.0%
II	3.0	4.36	6.0	7.0%
I	5.0	7.44	30	10.0%

2.6 Multivariate Models

Multivariate models were developed in an attempt to determine which factors present in the software development process have a significant effect on the quality and reliability of the delivered product. One such model, the Rome Laboratory Prediction Model RL - TR-92-52, was developed to predict fault density at delivery time. It also predicts the total number of faults, N, and the failure rate λ. The unique feature of this model is that it incorporates a wide range of factors, including software characteristics, the development environment and the application type. The various factors are listed in Table 4.

Table 4. TR-92-52 Factors

TERMS	DESCRIPTION
A	Factor selected based on Application type; represents the baseline fault density
D	Factor selected to reflect the Development environment
S	Factor calculated from various "sub-factors" to reflect the Software characteristics
SLOC	The number of executable Source Lines Of Code (non-blank, non-comment)
FD	Fault Density; defined as the ratio of faults to SLOC
N	Estimate of number of faults in the system, derived from fault density and SLOC
C	Factor representing a Conversion ratio for each Application type; determined by dividing the average operational failure rate by the average fault density for each type

This is one of the few publicly available prediction models based on extensive historical data. The data were obtained from a wide range of software systems that were developed for the Air Force, including airborne, strategic, tactical, process control, production center and development, and ranging in size from 88,000 LOC to 2,500,000 LOC.

There are many more software reliability models available, each with a slightly different perspective. Please see [8, 10, 11, 15, 21] for further information. No model has yet been found that is applicable in all situations.

3. Validation of Life-Critical Software

Validation of ultra-reliable software is a prohibitively expensive and time-consuming effort. As mentioned in Section 1, Butler and Finelli [1] provide an analysis of the problems associated with validating the reliability of software systems at the ultra-reliable level. The discussion that follows is drawn from that work.

To prove (validate) that a software program meets the requirement for ultra-reliable operation given in Table 1, it is necessary to operate the program for at least 10^7 hours without a failure. Assuming that the program runs continuously (i.e., 24 hours per day, 7 days per week) this would take 1,141.5 years. The impracticality of this is obvious. Faced with this reality, Littlewood [13] writes: "*Clearly, the reliability growth techniques . . . .are useless in the face of such ultrahigh reliability requirements.*" This is the underlying reason for the universal use of reliability growth models during testing.

Various techniques have been employed to overcome this problem, including the concept of fault tolerant design. But while fault tolerant systems do increase the level of confidence in the product (the likelihood that two functionally identical programs written by independent developers will fail at the same point is very low), they do not solve the validation problem. The two systems would require at least 10^7 hours of continuous defect-free operation to be validated. By the same reasoning, it can be seen that software reliability models likewise cannot be validated at the ultra-reliability level. The unsettling conclusion is that we have no feasible way of proving that software designed for ultra-reliable applications is actually that reliable. Therefore, we focus on a static model that uses a proven design metrics technology to gauge software reliability.

4. The Design Metrics

The Software Engineering Research Center (SERC) was established in 1986 as a National Science Foundation Industry/University Cooperative Research Center. The Design Metrics Research Team working in SERC developed a set of metrics to aid in the identification of defect-prone components in software under development. Created from empirical studies of actual defects in a wide range of software programs, the metrics were based on information available during the design phase of software development. In subsequent studies, researchers were able to create a model based on the design metrics that predicted which software components were likely to be defect-prone after coding. A number of studies by independent researchers have since validated the design metrics model and its ability to identify defect-prone components during the design phase.

4.1 The External Design Metric, D_e

The first metric, D_e, is a measure of the external relationships the component has to other components in the software system. This information is available during the preliminary design (architectural design) phase of software development, and reflects the component's fan-in and fan-out and the in-flow and out-flow of data through the component. D_e is calculated from:

$$D_e = e_1 \text{ (inflows * outflows)} + e_2 \text{ (fan-in * fan-out)}$$

where

inflows, outflows = the number of data entities passed into and out-of the component

fan-in, fan-out = the number of superordinate and subordinate components directly connected to the component.

e_1, e_2 = weighting factors.

The term (inflows * outflows) provides an indication of the data flow through the component, while the term (fan-in * fan-out) gives the number of invocation sequences through it. These terms are of degree two because research showed this to provide the best fit with the observed data.

4.2 The Internal Design Metric, D_i

The second metric, D_i, is a measure reflecting three areas internal to a component which research indicated were major sources of defects. This information is available during the detailed design phase of software development, and reflects the component's internal structure. D_i is calculated from:

$$D_i = i_1 \, CC + i_2 \, DSM + i_3 \, I/O$$

where CC = Central Calls, the number of procedure or function invocations

DSM = Data Structure Manipulations, the number of references to complex data types

I/O = Input/Output, the number of accesses to external devices

i_1, i_2, i_3 = weighting factors.

4.3 The Composite Design Metric, D(G)

The research team developed a third metric, D(G), which is the simple sum of D_e and D_i.

$$D(G) = D_e + D_i$$

With this metric, researchers were able to account for decisions throughout the design stage that could affect the quality of the delivered product.

4.4 Experimental Results

The research team undertook studies to determine the effectiveness of the three metrics in identifying defect-prone components. On student projects using assorted programming languages, and varying in length from 2,000 to 30,000 LOC [25], D_e identified 12% of the components as defect-prone. Half of these actually contained defects and more than half (53%) of all of the detected defects in the programs were in the identified defect-prone group.

When the same programs were used to study the effectiveness of D_i, the team found that 89% of the identified components actually contained defects and that 94% of all of the detected

defects in the system were contained in those components. Only 11% of the components were falsely identified as defect-prone.

The best results were obtained using the third metric, D(G). On the student projects [25], 100% of the components identified as defect-prone by D(G) actually contained defects, and 97% of all of the detected defects in the programs were contained in the identified group. No components were falsely identified as defect-prone.

4.5 Comparison to Other Metrics

In addition to studying the effectiveness of the design metrics, D_e, D_i and D(G), the research team compared them with the time-honored McCabe's cyclomatic complexity metric, V(G), and with the lines-of-code metric, LOC [9]. Results of the study are shown in Table 5.

Table 5. Comparative Effectiveness of Defect-Detecting Metrics

CATEGORY	V(G)	LOC	D_e	D_i	D(G)
Components Highlighted	11%	11%	12%	11%	12%
Highlighted Components with Defects	44%	56%	50%	89%	100%
Detected Defects Found	37%	51%	53%	94%	97%
False Positives	56%	44%	50%	11%	0%

In the study [25], a component was identified as defect-prone if the V(G) value was greater than 10, which is the value chosen by McCabe as being a reasonable upper limit. For the LOC metric, a component was identified if the LOC value was in the upper 11% of all the component LOC values in the study. Note how well D_i performed against both V(G) and LOC in all three categories. This performance is even more significant considering the fact that D_i is available during the early part of the design phase (architectural design), sooner than either V(G) or LOC.

The performance of D(G) is even more startling. When compared to V(G) and LOC, not only did every component identified by D(G) actually contain defects, but also 97% of all the defects in the programs studied were contained within those identified components. Further, no false positives were identified by D(G) as compared to 56% false positives for V(G) and 44% for LOC. As is the case for D_i, the D(G) metric is available during the design phase of software development.

Numerous other studies have been conducted to validate the applicability of the design metrics in identifying defect-prone components in large-scale industrial software. These studies vary widely in terms of the implementation language and the application domain. In all such studies, these metrics have proven to be an effective approach in targeting defect-prone software components [22, 26, 27].

5. Empirical Reliability Study

The ability of the design metrics to accurately identify defect-prone components in a software system prior to the start of coding has been so effective that the question was asked whether they might also be used to gauge the reliability of the delivered product. Obviously, the availability of such a gauge during the design phase would be of significant value to a developer. Accordingly, a study of a large, complex software product was undertaken to determine whether such a relationship existed and, if so, what its nature might be. The first thrust of the study, reported here, was confined to a subset of a large-scale program, with the goal of identifying such a relationship, developing and quantifying the relationship hypothesis to a first approximation, and determining the feasibility and direction of further research.

5.1 The Target System

The software program selected for the study was a large defense program. It consists of approximately 1,500,000 lines of Ada code and was developed over a period of several years by a group of about 200 software engineers. New releases of the product, each with additional functionality, are developed according to a well-defined schedule. Problems identified in the field are documented, categorized and prioritized. For this study, only those problems categorized as ‘defect’ and having priority 1 or 2 out of 4, where 1 and 2 are most significant, were chosen. This eliminated cosmetic problems and minor defects of little significance (as determined by the customer). The study was confined to 956 individual procedures, functions and packages, and their associated defect reports.

5.2 The Data

To aid in the collection of the design metrics, a Design Metrics Analyzer for Ada (DMAA) was used. The DMAA tool not only sped up metrics collection, but also provided consistency in calculating metric values. The data consisted of 956 packages, procedures and functions contained in a total of 46 files. The files were analyzed with the DMAA tool, which

calculated the values of D_e, D_i and D(G) for each of the packages, procedures and functions within each file. From the problem reports returned by the customer, the number of defects associated with each file was determined. Because the problem reports did not identify which particular procedure or function within a file actually contained the defect, the average value of D_e, D_i and D(G) was calculated for each file and used in the study. Thus, each file had an associated defect count and an average value of D_e, D_i and D(G). The average was chosen to eliminate the problem of a large file containing many procedures or functions, each having small values for the metrics, outweighing a smaller file having fewer but larger metric values.

5.3 Experimental Results

The data were sorted in ascending order of the D(G) average for each file and the corresponding value of Defect-Count was plotted as the dependent variable. A best-fit trend line was computed for the dependent variable, Defect-Count, and plotted. A positive relationship between Defect-Count and the design metric, D(G), is clearly visible, as shown in Figure 1.

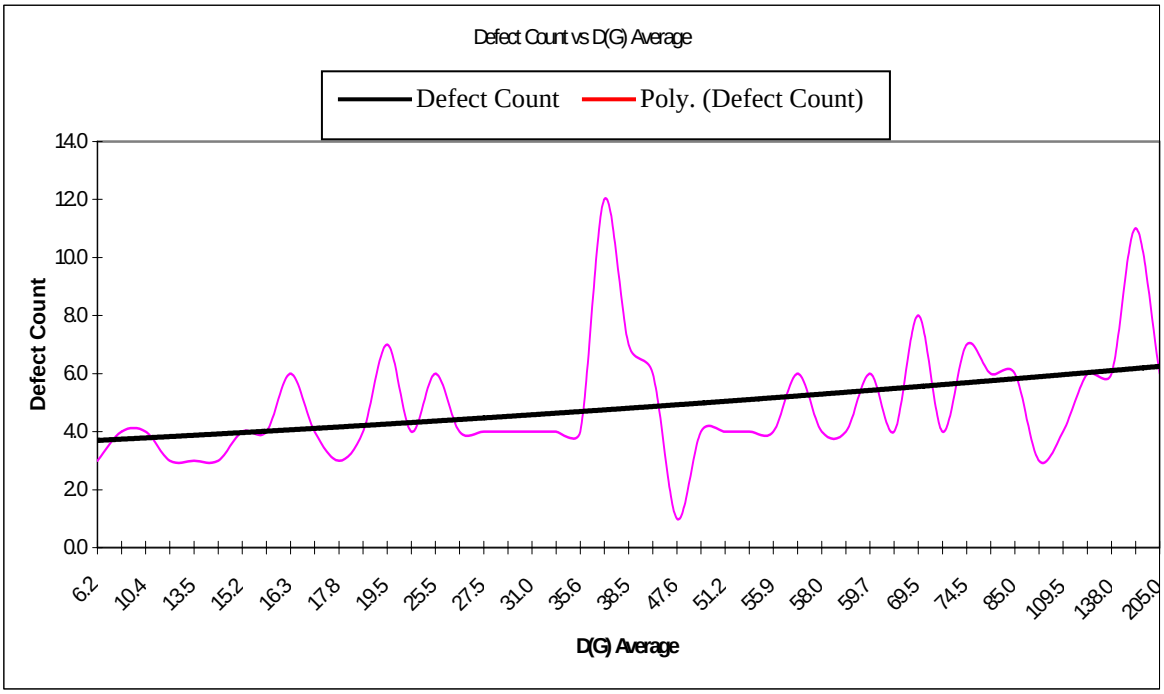

Figure 1: Trend line with Defect Count vs. D(G) Average

The trend line is a second-order polynomial and is given by the expression:

$$N_d = 0.0003D(G)^2 + 0.0422D(G) + 3.6567$$

where N_d is the number of estimated defects (defect count), and D(G) is the design metric introduced in Section 4.3. Thus, a first approximation to the relationship between the design metric, D(G), for a file and the number of defects that file will have in the field has emerged. Furthermore, if N_d is the number of defects in a file and n is the number of files in a product

then clearly $$N_p = \sum_{i=1}^{n} Nd_i$$

where N_p is the total number of defects in the product. Using the definition of the reliability of a software product as introduced in Section 1 produces

$$R_e = 1 - (N_p / LOC).$$

Substituting for N_p in terms of D(G) results in

$$R = 1 - \left[\frac{\sum_{i=1}^{n} \left(0.0003 D(G)_i^2 + 0.0422 D(G)_i + 3.6567 \right)}{LOC} \right]$$

which is the first approximation for the expected reliability of a software product in terms of the design metric, D(G).

6. Design Significance and Software Stress

The relationship between the expected reliability (R_e) and the composite design metric, D(G), for a single software component (n = 1) can be expressed in the general case as:

$$R_e = 1 - \left[\frac{aD(G)^2 + bD(G) + C}{LOC} \right]$$

where C is the process latent defect capability in defects at delivery, based on the process maturity level. We define the *design significance D(S)* of a software system S as

$$D(S) = [aD(G)^2 + bD(G)] / C$$

Note that D(S) incorporates the components of R_e affected by design decisions. In general, the point at which the design decisions reflected in D(G) become significant to the software component reliability occurs when D(S) = 1. At this point the effect of design decisions on the expected reliability of the software is as significant as the process defect capability, C. When D(S) < 1, the effect of design decisions on the expected reliability of the software is less significant than C; when D(S) > 1, the effect of design decisions on the expected reliability of the software is more significant than C. In practice the project engineer or design team may set the point of significance of D(S) to a higher or lower value, depending on the project reliability requirements. Thus the expected reliability may be designed into the software, subject to the process capability.

The relationship between CMM process maturity level and the corresponding defect density is displayed in Table 3. As the software development process becomes more mature in terms of the CMM rating, the latent defects decrease (as seen in the defect density data in Table 3), thus making the inherent design decisions reflected in D(G) more significant. For example, in a CMM Level 1 process, the design decisions become significant when $aD(G)^2+bD(G) \sim 5.0$ (Keene data) and for a CMM Level 5 process, the design decisions become significant when $aD(G)^2+bD(G) \sim 0.5$. For a 1KLOC software component this occurs when D(G) is approximately 80 for a CMM Level 1 process and approximately 11 for a CMM level 5 process. Actual D(G) values in studies have ranged from 2 to more than 19,000.

Thus by calculating D(S) for each software component as it is being developed, it is possible to identify those components for which design decisions are significant to the reliability of the product. This measure is used to focus reliability improvement efforts to those components where it would be most effective. It is also possible to design a software component or program to have a specific software reliability parameter.

For a software component, stress is defined as the likelihood that it will fail (break) in operational use. Studies on D(G) and reported field failures indicate that D(G) is accurate in identifying components likely to fail in operation, in other words, components with significant stress. In addition, the value of D(G) is directly related to the magnitude of stress on the component. Therefore, by measuring D(G) it is possible to identify those components in a system which are the stress points (most likely to fail in the field, highest value of D(G)) while still in development and to modify the components to alleviate the stress or preclude such failure. Further discussion of stress points in software design can be found in [25, 26, 27].

7. High Reliability Engineering Process – In Practice

As a result of these advances, and in response to customer requests for specific reliability performance, a process was developed to identify the design significance D(S) points (the default is D(S) = 1), stress points, and expected reliability in the software as it is being developed and to measure the magnitude of these attributes at each D(S) point. Techniques were also developed to incorporate these parameters into the design of the product while still in development. Beginning with the adaptive reliability process used by Peterson [20] as a base, modifications were made to provide measurement and analysis of the stress, design significance and expected reliability parameters needed, and to close the feedback loop with the design team to allow the software to be improved. The process is depicted in Figure 2. Further information on the adaptive reliability process can be found in [16, 17, 18, 19].

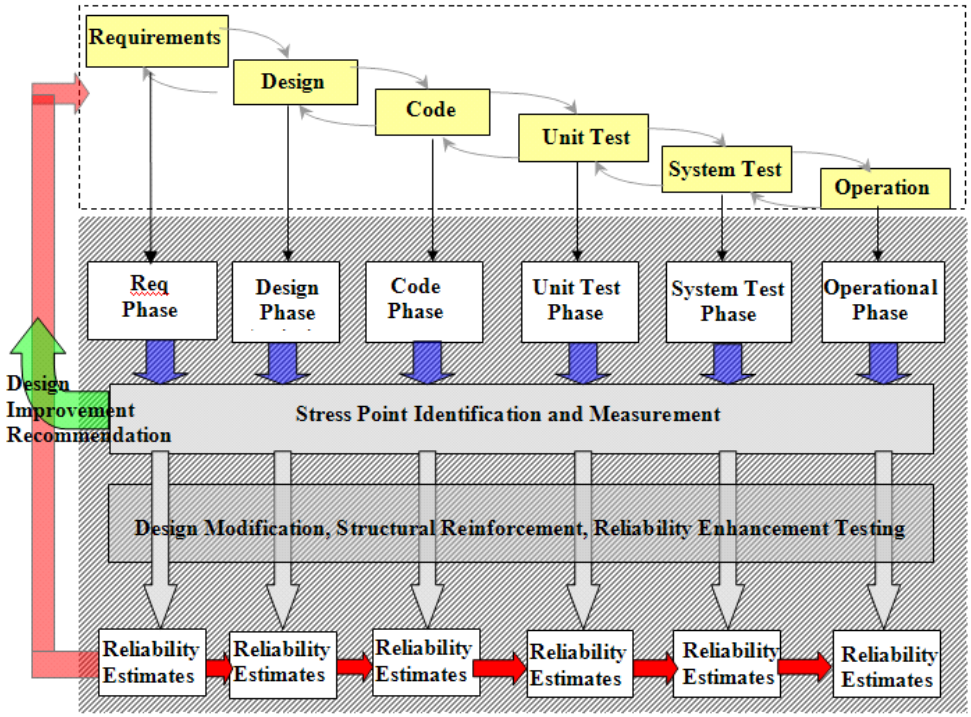

Figure 2: High Reliability Engineering Process

The result is a highly practical high reliability engineering process which incorporates the concepts and techniques described in this paper. This process is designed to be used at every stage of the software life-cycle and is currently being implemented in the code and test phases of an active development program. Research is underway to extend the process to earlier phases.

In the code phase an automated tool is used to measure the D(G) value of each component, followed by computation of D(S). From these data the components with the highest stress levels (top 12 – 15%) are identified, as well as those with the highest D(S). These

components are then analyzed with respect to the component parts of D(G), which allows the individual lines of code that are responsible for the stress or design significance within the component to be identified.

After the design significance, stress points and magnitudes are identified and measured, the process moves to a meeting with the design team where design modification alternatives are considered to alleviate the stresses and reliability concerns. Modifications typically include redesign to reduce data flows or fan-in/out, partitioning, procedure calls, data state manipulations or I/O statements - the primitives that form the composite D(G). The design modifications are implemented and, if warranted, the component reliability and stress parameters are re-measured. The extent to which a given component can or should be modified is always a trade-off between competing interests and the negotiated decision is arrived at by consensus.

The next step is structural reinforcement to strengthen the software in those places where design modification is not possible or where reinforcement is judged worthwhile. Reinforcement can take the form of parameter checking, assert statements, data verification, or other means. As noted before, the extent of reinforcement efforts is a trade-off.

Following structural reinforcement the software component is subjected to reliability enhancement testing. This is unit level testing in a high-stress environment that is specifically designed to break the component at the stress points. The reliability engineer works with the design and test teams in the development and application of these tests. The confidence in the ability of D(G) to accurately identify high stress components allows more focused and efficient allocation of expensive test resources to be used where they are most beneficial across the program. High stress components, as well as those identified as critical due to mission, safety, or other factors, are subjected to more strenuous testing than others. These components are tracked throughout the lifecycle and monitored to ensure that no degradation occurs to stress, reliability, or other parameter of interest.

8. Conclusion

The results of the study indicate that a positive relationship exists between the design metric, D(G), and the number of defects that are found in the product after it has been fielded. The relationship has been identified and quantified to a first approximation. Further, it has been shown that D(G) can be used to predict and measure the impact of design decisions on the expected reliability of a software product during the code and tests phases of development. The ability to identify and measure stress points in the software has also been shown. The concept of design significance was introduced and shown to be a viable and measurable attribute that can identify those components for which design decisions have a significant effect on the expected reliability of the software. These findings were sufficient to justify development and implementation of a high reliability engineering process.

A high reliability engineering process which incorporates the concepts and techniques described herein was developed and placed into practice on an active software development program requiring ultra-reliability. With this process it is possible to engineer reliability into the design of the product. The result is a significant improvement in the practice of software engineering and may be the subject of a future paper.

It must be noted that the accuracy of the findings is subject to the wide variations in the dependent variable, Defect-Count, with respect to the independent variable D(G), and to the large values in the standard deviation computed for the D(G) associated with each value of Defect-Count. These variations are due in large part to the variable nature of the software development process itself, i.e., experience of the developers, efficiency of the peer-review process in defect identification and removal, extent to which the process is followed when schedule pressure occurs, etc. As the software development process is brought more into control, in the statistical sense, these wide variations should diminish.

As Butler and Finelli pointed out so convincingly in [1], “neither this model nor any other can be validated for safety-critical real-time software applications. At best, using current state of the art hardware and software, validation can be accomplished only for low to moderate reliability applications.”

References

[1] Butler, R. and G. Finelli, "The Infeasibility of Quantifying the Reliability of Life-Critical Real-Time Software", *IEEE Trans. Software Eng*., Vol. 19, No. 1, Jan. 1993.

[2] Chen, H. M.R. Lyu and W.E. Wong, "An empirical study of the correlation between code coverage and reliability estimation," in Proceedings of the Third IEEE International Software Metrics Symposium, pp 133-144, Berlin, Germany, March 1996.

[3] Chen, H. M.R. Lyu and W.E. Wong, "Incorporating code coverage in the reliability estimation for fault-tolerant software," in the Proceedings of the 16th Symposium on Reliable Distributed Systems, pp 45-52, Durham, NC, October 1997.

[4] Diaz, M. and J. Sligo, "How software process improvement helped Motorola", *IEEE Software*, Vol.14, No.5, pp.75-81, 1997.

[5] Dietz, T. and N. Malik, "eQualite: Quality Assessment of Software Suppliers", Acquisition of Software-Intensive Systems Conference, Carnegie Mellon Software Engineering Institute, January 2003.

[6] Dunn, Robert H., Software Quality, Concepts and Plans, p. 36, Prentice Hall, 1990.

[7] Dyer, M., The Cleanroom approach to quality software development, Wiley, 1992

[8] Farr, W., "A Survey of Software Reliability Modeling and Estimation," Naval Surface Warfare Center, NSWC-TR-82-171, 1983.

[9] Gaffney, J., "Estimating the Number of Faults in Code", *IEEE Trans. Software Eng*., Vol.10, No.4, 1984.

[10] Jones, C., "The Pragmatics of Software Process Improvements". In the 'Software Engineering Technical Council Newsletter', Technical Council on Software Engineering, IEEE Computer Society, Vol. 14 No. 2,Winter 1996.

[11] Lennselius, B., "Software Complexity and its impact on software handling processes," *Proceedings of the Sixth International Conference on Software Engineering Telecommunications Switching Systems,* April, 1986.

[12] Lipow, M., "Number of Faults per Line of Code", *IEEE Trans. Software Eng*., Vol.8, No.4, 437-439, July, 1982.

[13] Littlewood, B., "Predicting Software Reliability," *Philosophical Transactions of the Royal Society*, London, 1991.

[14] Mathur, A.P. and S. Krishnamurthy, “On the Estimation of Reliability of a Software System Using Reliabilities of its Components”, *Proceedings of the 8th International Symposium on Software Reliability Estimation*, Albuquerque, NM, November 1997.

[15] Musa, J., Iannino, A., Okumoto, K., Software Reliability: Measurement Prediction, Application, McGraw-Hill, 1987.

[16] Peterson, J, and M. Yin, "Modeling Software Reliability by Applying the CASRE tool Suite to a Widely Distributed, Safety-Critical System" *Proceedings of the International Society of Software Reliability Engineers* (ISSRE) Conference, 2001.

[17] Peterson, J. "Modeling Software Reliability for a Widely Distributed, Safety-Critical System", *Reliability Review*, Volume 22, Number 1, pp 5-26, March 2002.

[18] Peterson, J. and S. Keene, "Analyzing the Impact of Specification Changes on Software Reliability", *Proceedings of the International Society of Software Reliability Engineers (ISSRE) Conference*, 2002.

[19] Peterson, J, "A Tool for Calculating and Organizing Software Reliability Predictions for a Complex System", Proceedings of the International Society of Software Reliability Engineers (ISSRE) Conference, 2003.

[20] Peterson, J., “The Historical Results and Lessons Learned of Software Reliability Prediction**,”** 4 th Raytheon Systems/Software/Processing Systems Engineering Symposium, April 5-7, 2005

[21] Putnam, L. and W. Myers, Measures for Excellence: Reliable Software on Time, within Budget, Prentice-Hall, 1992.

[22] Wong, E., J.R. Horgan, M. Syring, W. Zage, and D. Zage, "Applying Design Metrics to Predict Fault Proneness: A Case Study on a Large Scale Software System," *Software-Practice and Experience*, Vol. 30, No. 14, pp. 1587-1608, November 25, 2000.

[23] Yang, M., W.E. Wong and A. Pasquini, “Applying Testability to Reliability Estimation”, SERC Technical Report, 1999.

[24] Yin, M., L. James, S. Keene, R. Arellano, and J. Peterson, “An Adaptive Software Reliability Prediction Approach,” 23rd Annual Software Engineering Workshop, NASA/Goddard Space Flight Center, Greenbelt, Maryland, Dec 2-3, 1998.

[25] Zage, W. and D. Zage, “Relating Design Metrics to Software Quality: Some Empirical Results,” Software Engineering Research Center Technical Report SERC-TR-74-P, May 1990.

[26] Zage, W. and D. Zage, “Evaluating Design Metrics on Large Scale Software”, *IEEE Software*, Vol. 10, No. 4, pp. 75-81, July 1993.

[27] Zage, W., D. Zage, J.M. McGrew and N. Sood, “Using Design Metrics to Identify Error-Prone Components in SDL Designs”, *Proceedings of the 9th International SDL Forum*, Montreal, Canada, published by Elsevier Science, UK, June 1999.