Predicting Maintainability with Software Quality Metrics

Sallie Henry
and Steve Wake

TR 88-46

Predicting Maintainability with Software Quality Metrics

by

Sallie Henry
and
Steve Wake

Computer Science Department
Virginia Tech
Blacksburg, VA 24061
(703) 231-7584

Predicting Maintainability with Software Quality Metrics

Abstract

Maintenance of software makes up a large fraction of the time and money spent in the software life cycle. By reducing the need for maintenance these costs can also be reduced. Predicting where maintenance is likely to occur can help to reduce maintenance by prevention. This paper details a study of the use of software quality metrics to determine high complexity components in a software system. By the use of a history of maintenance done on a particular system, it is shown that a predictor equation can be developed to identify components which needed maintenance activities. This same equation can also be used to determine which components are likely to need maintenance in the future. Through the use of these predictions and software metric complexities it should be possible to reduce the likelihood of a component needing maintenance. This might be accomplished by reducing the complexity of that component through further decomposition. Even though this is only one study, this methodology of developing maintenance predictors could be applied in any environment.

I. Introduction

Computer scientists are continually attempting to improve software system development. Systems are developed in a top-down fashion for better modularity and understandability. Performance enhancements are implemented for more speed. One area in which a great deal of effort is being devoted is software maintenance. Brooks [1] estimates that fifty percent of the development cost of a software system is for maintenance activities. Since a large portion of the effort of a system is devoted to maintenance, it is reasonable to assume that driving down maintenance costs would drive down the overall cost of the system.

Measuring the complexity of a software system could aid in this attempt. By lowering the complexity of the system or of subsystems within the system, it may be possible to reduce the amount of maintenance necessary. Software complexity metrics were developed to measure the complexity of software systems. This study relates the complexity of the system as measured by software metrics to the amount of maintenance necessary to that system.

The Software Life Cycle

To better understand a software system, it is helpful to look at how a system is developed. Most software systems follow a software life cycle. Although some stages of the software life cycle may not be formally present, the basics of that stage of the life cycle are almost certainly used in the development of a software system.

Ramamoorthy divides the software life cycle into seven stages [2]. The first stage is the understanding of requirements and of the problem in question. During this stage, requirements are developed for the problem that is to be solved by this system, the functionality of the system and any constraints which are to be placed on the system.

Specification of requirements is the second stage of the software life cycle. During this stage, software specialists attempt to understand the requirements and develop a set of specifications of what the system is to do without describing how the system is to do it.

Design is the third stage of the software life cycle. The problem is decomposed into a number of modules and the ways in which they interact. Modules may be further broken down into submodules and procedures in order to obtain a unit that can be easily understood and programmed. The sum of all these units is the design of the system and should meet the specification from the previous stage and answer the question of how the system is to be implemented.

The design from the previous stage is then used in the fourth stage, implementation. In implementation, the modules are coded in a suitable language for the problem.

In the testing stage, the code is executed and bugs may be found and corrected before the system is released. These bugs may occur in the implementation, in the requirements or in the design. Individual modules are tested in this stage along with interactions between modules and, finally, the system as a whole is integrated.

The last two stages are maintenance and evolution. These two stages are similar in that there are changes being made to the system. If bugs are found during the operation of the system, this stage is maintenance, but if performance improvements are made, it could be either evolution or maintenance. The boundary between what is evolution and what is maintenance is usually decided by how major a change is to the system. A major change would be evolution and a minor change is maintenance. Since it is a subjective opinion as to what is major, the distinction between the two phases can be difficult.

There are various other software life cycles proposed in the literature. The model proposed by Bennington [3] is the stagewise model. It proposes nine stages for software development: operational plan, machine and operational specifications, program specifications, coding specifications, coding, parameter testing, assembly testing, shakedown, and system evaluation. Boehm [4] suggests that software be developed as a sequence of tasks waterfalling into one another. These tasks are: system requirements, software requirements, preliminary design, detailed design, code and debug, test and preoperations, and operations and maintenance. Each step includes validation and verification activities. Balzer [5] proposes a life cycle model for automatic programming. This model includes steps for requirements analysis, validation, maintenance, mechanical optimization, and tuning. Since maintenance is a critical phase in the software development process, all of these software life cycle models include a maintenance phase.

Software Maintenance

Software maintenance activity is a major part of the software life cycle. Estimates of time and money spent on this stage of the software life cycle range from forty to sixty-seven percent of the total for the entire life cycle [2][6]. Lientz [7] suggests that "maintenance and enhancement tend to be viewed by management as at least somewhat more important than new application software development." Curtis [8] states that "more time is spent maintaining existing software than developing new code." Since "the cost of correcting program errors can (and typically does) increase enormously with time to discovery" [9], it is important to find these errors as early as possible.

Swanson [10] characterizes three types of maintenance activities. They are corrective maintenance which is performed in response to a failure, adaptive maintenance which is performed in anticipation of a change within the environment of the system, and perfective maintenance which is performed to enhance maintainability. Each of these types of maintenance is an important part of the maintenance process. However, in this study we are looking at corrective maintenance of a software system. By determining where errors occur, we hope to be able to predict where future errors might occur. This allows preventative maintenance to be done to minimize the amount of future corrective maintenance work and makes software systems more sound.

Ramamoorthy [2] describes three ways to reduce maintenance costs: "(1) The system must be developed with maintenance in mind; (2) The system must be maintained with future maintenance in mind; and (3) The system must be continually upgraded to cope with future technology." He suggests that the use of precise methodologies would solve many maintenance problems. One such method would be formal proofs of correctness. However, proving a program correct is a time consuming process which is sometimes difficult to verify. Another possible method is testing. Since exhaustive testing is not possible, test cases must be carefully designed to cover the entire program. This is generally not possible, therefore bugs can penetrate the code.

Software Quality Metrics and Maintenance

One tool which helps in solving some of the problems of software maintenance is software quality metrics. The metrics quantitatively measure aspects of the system which can be used as indications of the quality of the software system. Metrics can be used at various stages of the life cycle. Ramamoorthy [2] suggests that metrics can be used for maintenance purposes during the requirements, implementation, testing, and maintenance stages. In this study, we view the software quality metrics as a tool used in the maintenance phase of the software life cycle.

Yau and Collofello [6][11] have developed a software metric to measure the ripple effect of modifications in a software system. The ripple effect is "the phenomena by which changes to one program area have tendencies to be felt in other program areas." If the ripple effect is large, a modification to one module of a system may have impact on many other modules in the system. This leads to high maintenance costs and low system reliability.

Basili [12] has attempted to determine the correspondence between the software science measures of Halstead [13] and other related metrics to the number of development errors and to the weighted sum of effort required to isolate and fix these errors on a number of FORTRAN projects. Most of the correlations are weak, but this is attributed to the discrete nature of error reporting and to the fact that most of the modules examined reported zero errors. We are attempting a similar study.

Henry and Kafura [14] used the information flow metric to analyze the UNIX operating system. They chose UNIX for several reasons including it has large enough size and the fact that it is not a toy or experimental system but a system designed for users. They found that a high complexity shows stress points in the system or inadequate refinement of the procedure. By correlating changes in the system with the complexity of the procedures they found a high correspondence between these values.

Kafura and Reddy [15] studied the use of software complexity metrics on several versions of the same software system. The system they studied was a data base management system developed by students at Virginia Tech over a number of years. It is a medium size software system (16,000 lines) written in FORTRAN. They decided to use a subjective evaluation technique in order to determine whether software metrics could provide information to a maintainer of a system in order to avoid poorly performed maintenance. Subjective evaluation means that they "attempt to relate the quantitative measures defined by the software metrics to the informed judgement of experts who are intimately familiar with the system being studied." An important part of their investigation was examining the changes in the system from one version to the next. They found that the change in the complexities of the software metrics agreed with the changes that one would expect from the changes in the software system. Another interesting finding was that there was a growth in structural complexity as a result of maintenance activity. Two possible uses for software metrics in the maintenance process were suggested. "First, the metrics can be used to identify improper integration of enhancements. ... Second, procedures which are perceived to be complex can lead to improper structuring of the system because maintainers will avoid dealing with this complex procedure when making enhancements, even when the maintainer knows that a major restructuring of the complex component is called for in order to gracefully include the required enhancements."

Section II describes the metrics and the metric tool used in this research. The data used in this experiment is briefly discussed in section III. While predicting maintainability does not seem trivial, the paradigm we developed to generate equations to predict maintenance activity is presented in section IV. Our conclusions are given in section V.

II. Software Metrics

Software maintenance occurs because software does not do what it was designed to do. Higher quality software is less likely to need corrective maintenance. Higher quality software is less likely to need corrective maintenance. However, quality is a subjective term. If there is to be an improvement in the quality of software, there must be a way to objectively, or quantitatively, measure quality. This is the realm of software quality metrics. Software quality metrics provide a way to quantitatively measure the quality of software. These metric values can then be used as indicators to determine which portions are likely to require maintenance.

There are three classifications of metrics that are used to measure the quality of source code: *code metrics* which measure physical characteristics of the software, such as length or number of tokens; *structure metrics* which measure the connectivity of the software, such as the flow of information through the program and flow of control; and *hybrid metrics* which are a combination of code and structure metrics. The metrics are briefly discussed in this section. Interested readers are asked to refer to the references for more details.

Code Metrics

Historically, these were the first metrics and are among the simpler metrics to determine. These metrics include Length (measured in lines of code), McCabe's Cyclomatic Complexity and Halstead's Software Science Indicators.

A line of code, or length, is any line of program text that is not a comment or a blank line regardless of the number of statements or fragments of statements on the line. This specifically includes all lines containing program headers, declarations, and executable and non-executable statements [16].

McCabe's Cyclomatic Complexity, denoted V(G), was designed to measure the number of distinct paths through a particular program by representing the program with a graph and counting the number of nodes and edges [17]. The cyclomatic complexity for a graph with e edges and n nodes is:

$$V(G) = e - n + 2$$

One of the most commonly used metrics, Halstead's Software Science Indicators [18][19][20][21][22] are our third set of code metrics. Halstead considered a computer program as a collection of tokens which can be classified as either operands or operators. From these measures he developed a number of metrics giving an indication of the complexity of the program [13]. The basic measures are:

n1	=	the number of unique operators
n2	=	the number of unique operands
N1	=	the total occurrences of operators
N2	=	the total occurrences of operands

The size of a program, N, expressed in tokens, is:

$$N = N1 + N2$$

Vocabulary is defined as:

$$n = n1 + n2$$

These two measures lead to a third measure which Halstead calls volume:

$$V = N \times \log_2 (n)$$

Programming effort is a predictor of the effort it takes a programmer to translate ideas about a program solution into the implementation of that solution in a language known to the programmer. The formula for the effort indicator is:

$$E = \frac{V}{\frac{2}{n1} \times \frac{n2}{N2}}$$

Structure Metrics

In order to measure the complexity of a procedure with respect to its environment, Henry and Kafura developed the Information Flow Metric [14]. This metric attempts to measure the complexity of the code due to the flow of information from one procedure to another. Flows of information into a routine are called *fan-ins* and flows of information out of a routine are called *fan-outs*. A more formal definition for each is:

fan-in — the number of local flows into a procedure plus the number of global data structures from which a procedure retrieves information

fan-out — the number of local flows from a procedure plus the number of global data structures which the procedure updates

Local flows represent the flow of information to or from a routine through the use of parameters and return values from function calls. Combining these with the accesses to global data structures gives all possible flows into or out of a procedure. The complexity of a procedure p is defined as:

$$C_p = (\text{fan-in} \times \text{fan-out})^2$$

Hybrid Metrics

Hybrid metrics combine aspects of code and structure metrics. Two hybrid metrics are used in this research. Woodfield's Syntactic Interconnection Model is a hybrid metric which attempts to relate programming effort to time [23]. Woodfield defines a connection relationship between modules A and B. A connection relationship is a partial ordering between modules A and B such that one must understand the function of B before one can understand the function of A.

There are three types of module connections: *control*, *data* and *implicit*. A control connection implies an invocation of one module by another. A data connection occurs when a module uses a variable modified by another module. An implicit connection occurs when there are some assumptions used in one module that are also used in another module. One example is that two modules may both make the assumption that input is an expression of eighty characters or less. If this changes then both modules have to be modified to reflect that change.

The connection A -> B implies that some aspect of module B must be reviewed and understood before module A is completely understood. The number of times a module must be reviewed is Woodfield's definition of the module's fan-in. He presents the following general equation for the model:

$$C_b = C_{1b} \times \sum_{k=2}^{\text{fan-in}-1} RC^{k-1}$$

where

C_b = the complexity of module B's code

C_{1b} = the internal complexity of module B's code

fan-in = the sum of the control and data connections for B's code

RC = a review constant

The internal complexity for the module can be any code metric. In Woodfield's model definition, Halstead's Program Effort Metric was used. The model uses a review constant of 2/3 which is a number previously suggested by Halstead.

Henry and Kafura's Information Flow Metric can also be used as a hybrid metric. As a hybrid metric, the formula for the complexity is:

$$C_p = C_{ip} \times (\text{fan-in} \times \text{fan-out})^2$$

where

C_p = complexity of procedure p

C_{ip} = the internal complexity of procedure p

Fan-in and fan-out are as previously defined. C_{ip} may be any code metric's measure of procedure p's complexity.

Software Metric Analyzer

The software metric analyzer, shown in Figure 1, is a tool developed under the direction of Dr. Sallie Henry at Virginia Tech. Given the source code as input it calculates each of the metrics discussed previously.

Using the UNIX tools LEX and YACC along with a BNF grammar for the given language, the language dependent portion of the analyzer (pass 1), calculates the code metrics and translates the source code into an encrypted relation language. These code metrics are generated by pass 1 because they are counts generated using the original source code before it has been disguised in any way. The relation manager (pass 2), takes the relation language code from pass 1 and translates it into a series of relations which along with the code metrics from pass 1 are the input to pass 3 of the analyzer. Pass 3 uses the relations to calculate structure metrics [24]. Both the code metrics and the structure metrics can be combined to create the hybrid metrics. Pass 3 also displays the metrics in various groupings such as by metric or by procedure. Details of the encoding algorithm and the implementation of the tool can be found in [25].

III. The Experiment

The software system used for this experiment is Version 2.0 of an actual product consisting of 193 procedures comprising about 15,000 lines of C code (including comment lines and blank lines). A major computer vendor currently markets this product. The project is composed of a number of modules, each with a separate function. Each module is composed of one or more procedures having a like function such as all the string handling routines or all the parsing routines.

The modules were processed by pass 1 of the metric analyzer separately, at the vendor's site, generating a file containing the values for the code metrics and a separate file of intermediate code for the procedures in the module. Recall from the previous section that the code metrics include lines of code, values for Halstead's Software Science indicators and McCabe's Cyclomatic Complexity. Relation language was then processed to generate the structure and hybrid metrics: Woodfield, Information Flow, Information Flow with Length, and Information Flow with Effort.

In order to verify the interpretation of the metric numbers generated, there must be control data against which the interpretation can be tested. Where software metrics are a guide to maintenance of a software product, it is useful to see what changes are necessary to the product after a major release. Since the last major version of this product was version 2.0, the version which was measured, any modifications to the source code after this time are for maintenance reasons. This could be for bug fixes or performance improvements but is not new development of any kind. This data was obtained from a code library which was used in the development and maintenance of the product.

The Code Library

A code library was used to monitor accesses to the different modules of the product. All source code in a code library could be accessed by the code librarian program. When a bug is found by a customer, a software report is sent to the maintainers of the product who determine what changes, if any, are necessary. Any changes made to the source code are done through the use of the code librarian. As used in their development and maintenance strategy, a module of source code is checked out of the code library when changes are to occur. After the change is made, tested, and found to be correct, the changed module is checked back into the code library. Each time new changes are to be made to the code, the module involved must be checked out of the library and the corrected version checked in. This enables an automated history to be kept of accesses to a module of code.

The smallest unit of change is the line. A line can be either added to or deleted from a module. A modification to a line is therefore treated as a deleted line followed by an added line in the same place. For purposes of verification, any changes or modifications described are based on the changing of a line of the code. Groups of lines that are all changed at the same time can also be determined.

It is possible to make a complete listing of the source code with the lines of code added or deleted flagged to show when they were added and deleted. A routine was written to find these flagged lines in the source code and count the number of additions and deletions and where they occurred. By coupling this with a routine to determine which procedure the changes occurred in, a count of the number of changes, the number of lines changed, and how they were changed is obtained. These numbers can serve as control data for the interpretation of the metric numbers as they apply to the maintenance phase of the software life cycle.

Since it was not possible to measure the amount of time spent on each correction, we measure the amount of maintenance by the number of times the procedure was changed and the total number of lines changed for each procedure. The number of code changes shows the number of bugs in each procedure and the number of lines of code changed determines the size of the errors.

By tabulating the results of these data collection routines, a list is created of all the routines along with the corresponding number of lines added after the s release, number of lines deleted, and number of times these changes were made to each routine in the program. This gives an indication of the maintenance activity which occurred to the program.

IV. Maintenance Predictions

This section presents the methodology used in developing maintenance prediction equations using the metrics and the results of the statistical analysis of the data collected. Interrelationships among the various metrics and the changes to the code are shown. A discussion is presented on the various statistics used and the multiple regression model is also provided. A statistical analysis of the data is presented using the multiple regression model for both number of changes and number of lines of code changed. Finally, we select one of the models and demonstrate that it is a good predictor for the data that has been analyzed.

Intermetric Results

First, recall from the previous section the various metrics used in this study: Length, Halstead's N, V, E, McCabe's Cyclomatic Complexity – V(G), Woodfield's complexity, Information Flow, Information Flow with Length, and Information Flow with Effort. Statistical correlations among the metrics are revealed in Table 1. Notice that there is a high degree of correlation among the code metrics. This occurs because the code metrics are attempting to measure the same aspect of the code. However, there are low correlations among the code metrics, the hybrid metrics and the structure metric. This shows that the code, hybrid, and structure metrics are measuring a different aspect of the code. These correlation results were expected and they agree with other studies in the software metrics area [26][27].

Table 1. Intermetric Correlations

	Length	N	V	E	V(G)	Woodfield	Info-L	Info-E	Info
Length	1.000								
N	0.842	1.000							
V	0.973	0.862	1.000						
E	0.740	0.370	0.758	1.000					
McCabe	0.840	0.762	0.770	0.420	1.000				
Woodfield	0.436	0.485	0.434	0.215	0.310	1.000			
Info-L	0.065	0.110	0.067	0.011	0.022	0.088	1.000		
Info-E	0.138	0.170	0.158	0.103	0.091	0.113	0.838	1.000	
Info	-0.077	-0.062	-0.068	-0.049	-0.093	-0.051	0.830	0.502	1.000

Statistical Analysis

The goal of this study is to develop a model which uses metric values as parameters in order to predict the number of lines of code which will be changed and the total number of changes during the maintenance phase of the software life cycle. Lines of code changed and number of changes are the dependent variables in the statistical model and the metric values are the independent variables.

Our first attempt in developing a model to predict maintainability involved using a single independent variable (in this case a single metric) to statistically determine the prediction model. Although the results of this research are interesting, we felt that a single metric may not adequately determine a good predictor in all cases. Since we have shown that different metrics measure different aspects of the source code, a greater degree of accuracy may be obtained by using more than one of the metrics available .

In some cases a linear relationship is not present between a single independent variable and the dependent variable. In these cases it is better to express the model as a multiple regression model. This indicates that more than one independent variable has some bearing on the value of the dependent variable. Looking at correlations may not be enough to determine which variables are needed in the model since there may be some interactions among two or more independent variables which better explain the observed effects. In our case, it may be that a combination of structure, code and hybrid metrics better explain the variation in the number of lines of code changed (NLC) and the number of code changes (NCC).

There are various statistics available to help calculate the best multiple regression model. These include the PRESS statistic, MSE (mean squared error), and Mallow's Cp. The best fitting model should have a Cp approximately equal to the number of independent variables and low values for both PRESS and MSE [28].

The PRESS (Prediction Sum of Squares) statistic [28] is a good statistic for a predictive model because it takes into account the effects of fitting one value out of a set of data to the model specified by the rest of the data. Consider a set of N data observations. To calculate the PRESS statistic we first withhold the first data observation and calculate the coefficients for the model with the other N - 1 observations. This is repeated for each of the N data observations resulting in N prediction errors or residuals. The sum of the N residuals gives us a value for the PRESS statistic. Thus, the model with the lowest PRESS value is a candidate for the best model.

Mean Squared Error (MSE) takes into account the error from the predicted value and the actual value. A model is fit to the observations and an expected value is calculated for each independent variable based on the dependent variables associated with it. The difference between the actual value and the predicted value is squared. This is done for all N observations and the sum of these values is added and a mean calculated. Again, the model with the lowest MSE is a good candidate for the best model. Mallows' Cp [28] takes into account the fact that there is a prediction variance in the MSE statistic. This variance, summed over the N data observations is equal to the number of independent variables in the model. Thus, the best Cp is the one with the value close to the number of parameters in the model.

The observations were randomly divided into two groups, one with three fourths of the data and the other with the other fourth of the data. The large group is used to determine the predictive model and the small group is used to verify the model. Only the large group of data is used in the determination of the model so that the smaller dataset has unknown values which would be predicted. Since each of the statistics can give a different "best" model, there is only an indication of what the best models are. A set of best models can then be chosen and the unused data can be used to select the best model of that group. By calculating the sum of squared errors for each model using the unused data a "best" model is chosen.

Tables 2, 3, and 4 show the top models as selected by each of the three statistics for equations for the predicted number of lines of code changed. Table five shows the best candidate models for the predictor equation taking into account values for all three statistics. Tables six, seven and eight show the best model equations for the number of code changes. Table nine shows the best overall equations taking all statistics into account. Although the model using N, V, and E is chosen in this group, we chose not to use it because of collinearity among the three variables. Collinearity among the independent variables can cause the coefficients of the model to not accurately reflect te relative effects of trhe different variables.

Table 2. Top 5 Models Selected by PRESS Statistic

NLC = 0.42997221 + 0.000050156 E - 0.000000199210 INFO-E

NLC = 0.45087158 + 0.000049895 E - 0.000173851 INFO-L

NLC = 0.60631548 + 0.000050843 E - 0.000029819 WOOD - 0.000000177341 INFO-E

NLC = 0.33675906 + 0.000049889 E

NLC = 0.62562353 + 0.000050633 E - 0.000030739 WOOD - 0.000147075 INFO-L

Table 3. Top 5 Models Selected by MSE

NLC = 1.27935618 + 0.05500043 L - 0.001333387 V + 0.000054797 E - 0.11960695 V(G) - 0.000000142938 INFO-E

NLC = 0.42997221 + 0.000050156 E - 0.000000199210 INFO-E

NLC = 1.2782025 + 0.05693335 L - 0.001428534 V + 0.000054898 E - 0.11900135 V(G)

NLC = 1.30521150 + 0.06024787 L - 0.001438433 V + 0.000054545 E - 0.12321067 V(G) - 0.000163532 INFO-L

NLC = 1.53080447 - 0.000355426 V + 0.000056495 E - 0.08419100 V(G) - 0.0000001493221 INFO-E

Table 4. Top 5 Models Selected by C_p

NLC = 1.47735619 + 0.000054638 E - 0.10017668 V(G) - 0.0000067303 WOOD

NLC = 1.51830192 + 0.000054724 E - 0.10084685 V(G) - 0.000000158757 INFO-E

NLC = 1.57295997 + 0.000054615 E - 0.10037670 V(G) - 0.0000051632 WOOD - 0.000157250 INFO-L

NLC = 1.45518829 + 0.00005456 E - 0.10199539 V(G)

NLC = 1.57353731 - 0.002446765 N + 0.000054672 E - 0.08863879 V(G)

Table 5. Best Overall Candidate Models

NLC = 0.42997221 + 0.000050156 E- 0.000000199210 INFO-E

NLC = 0.45087158 + 0.000049895 E - 0.000173851 INFO-L

NLC = 0.60631548 + 0.000050843 E - 0.000029819 WOOD - 0.000000177341 INFO-E

NLC = 0.33675906 + 0.000049889 E

NLC = 1.51830192 + 0.000054724 E - 0.10084685 V(G) - 0.000000161798 INFO-E

NLC = 1.45518829 + 0.00005456 E - 0.10199539 V(G)

Table 6. Top 5 NCC Models Selected by PRESS statistic

NCC = 0.40552034 + 0.00001163 E - 0.000006267 WOOD - 0.0000000478 INFO-E

NCC = 0.38710077 + 0.00001158 E - 0.000006897 WOOD

NCC = 0.41056545 + 0.00001157 E - 0.000006517 WOOD - 0.000039367 INFO-L

NCC = 0.36846091 + 0.00001149 E - 0.00000005238 INFO-E

NCC = 0.34394979 + 0.000011418 E

Table 7. Top 5 NCC Models Selected by MSE

NCC = 0.25438629 + 0.0043307 N - 0.00062705 V + 0.00001471 E
- 0.0000000525 INFO-E

NCC = 0.17374520 + 0.0127384 L + 0.00369907 N - 0.00089433 V
+ 0.000014349 E

NCC = 0.25250119 + 0.0039729 N - 0.00059868 V + 0.00001454 E

NCC = 0.32020501 + 0.0136926 L - 0.00048185 V + 0.00001230 E

Table 8. Top 5 NCC Models Selected by Cp

NCC = 0.34394979 + 0.000011418 E
NCC = 0.41704985 - 0.000129275 V + 0.000012345 E
NCC = 0.36846091 + 0.000011488 E - 0.00000005238 INFO-E
NCC = 0.3871077 + 0.000011583 E - 0.0000068966 WOOD
NCC = 0.25250119 + 0.00397286 N - 0.00059868 V + 0.000014538 E

Table 9. Best Overall NCC Models

NCC = 0.34294979 + 0.000011418 E
NCC = 0.36846091 + 0.000011488 E - 0.00000005238 INFO-E
NCC = 0.38710077 + 0.000011583 E - 0.0000068966 WOOD
NCC = 0.25250119 + 0.003972857 N - 0.000598677 V + 0.000014538 E
NCC = 0.32020501 + 0.01369264 L - 0.000481846 V + 0.000012304 E

Prediction Example

Before using any model we do a complete residual analysis to verify that the model is apt. As an example, consider the model with the three independent variables E, V(G) and Info-E. This model was in the top 5 as selected by both Cp and MSE. It also did well in the PRESS statistic. This model might be selected as the best overall model due to it having the lowest sum of squared error and be used to predict the amount of maintenance to occur on a given procedure.

Table 10. Verification of Smaller Data Set

Actual Value	Predicted Value	95% Confidence Interval	
		Lower	Upper
3	3.8621	2.6045	5.1197
0	0.0688	-0.4065	0.5441
19	21.3017	17.5334	25.0700
2	1.3829	0.9753	1.7905
0	2.5266	2.1381	2.9150
0	1.2029	0.7557	1.6501
0	1.2029	0.7557	1.6501
0	1.2025	0.7553	1.6496
6	0.7059	0.3166	1.0951
0	1.2036	0.7989	1.6084
0	0.6292	0.2404	1.0181
3	1.1268	0.7137	1.5398
8	0.9211	0.5159	1.3263
0	-0.0369	-0.5060	0.4322
0	1.2620	0.8011	1.7229
0	0.4561	0.0711	0.8410
0	0.4178	-0.0187	0.8542
0	0.7993	0.4004	1.1982
4	1.4956	1.0353	1.9558
0	1.4176	1.0327	1.8024
3	-0.1131	-0.6075	0.3813
1	1.2117	0.8164	1.6071
2	0.8987	0.5004	1.2969
3	0.6932	0.2823	1.1041
0	1.4830	1.0705	1.8956
0	1.1632	0.7285	1.5978
0	1.2872	0.7031	1.8713
0	1.3403	0.9253	1.7552
0	1.2370	0.8023	1.6718
8	2.9747	2.3459	3.6035
0	1.0742	0.6856	1.4627
0	1.3950	0.9600	1.8300
4	1.1297	0.6970	1.5625
0	0.9291	0.5360	1.3222
0	1.2546	0.7937	1.7155
0	1.1428	0.7201	1.5655
0	1.3240	0.8892	1.7589
0	1.6340	1.2337	2.0343
0	1.0385	0.6511	1.4258
0	1.2822	0.8212	1.7431
2	1.1762	0.7880	1.5645
4	4.9560	4.5022	5.4097
2	1.1306	0.6970	1.5642
0	0.8634	0.4649	1.2618
0	0.9103	0.5121	1.3085
0	0.3713	-0.0209	0.7634
8	2.3896	1.5838	3.1955

To do this, the values for each of the metrics used in the model are calculated using the software analyzer discussed earlier. These values are then put in to the regression equation and a value for NLC is calculated. The following examples are actual measurements from two of the procedures analyzed. Consider values of 145,335 for E, 59 for V(G) and 1,308,016 for INFO-E. The prediction equation yields a value of 3.86 for NLC, the expected number of lines to change in this procedure. The actual number of lines changed in this procedure was 3. Using values of 12,246 for E, 21 for V(G) and 195,929 for INFO-E, the prediction equation yields a value of 0.07 for NLC. This procedure required no modifications. Table 10 shows that smaller (verification) set of data with actual values, predicted values, and 95% confidence intervals. The Pearson correlation between predicted values and actual values is 0.78. After verifying the model, the data was gathered todether to calculate a predictor equation for future values. The Pearson correlation coefficient for this "full" model improves to 0.94. The value calculated is not meant to be the value for the exact number of lines of code we expect to have to change but rather gives a good indication of what that value will be. If this is done for all procedures in the software system, a ranking of procedures in order of likelihood of maintenance can be determined. If this is done before the system is released, future maintenance may be prevented by changing or preferably re-designing the higher ranking procedures

Research of this type is necessary to reduce the cost of maintenance. A suggested use for this type of model is for an organization to first collect a significant amount of error or maintenance data. The second step is to fit the error data statistically to the models. This phase of model development creates models specific to the language, application, and environment area of the organization. A best set of models is found at this time.

These predictor values can be used during the coding, testing, and maintenance phases of development. The number of code changes (NCC) value shows the predicted number of errors in a procedure and the number of lines of code changed value gives an indication of the size of those changes.

During the coding phase the procedures with higher values for these predictors can be redesigned in order to reduce the complexity of the procedure. Remeasuring the entire system shows whether the changes made actually reduce the predicted maintenance. This process can be repeated until acceptable values are reached.

During the testing phase, procedures with higher valued predictors can be more thoroughly tested. Errors are then found and fixed before shipping which provides a better product. Errors are also less expensive to fix at this time.

After the product is shipped, if maintenance is necessary, test values for procedures may be used to help isolate the error. It is more likely that the error will be in a procedure with a very high value for predicted maintenance than a procedure with a low predicted maintenance value.

V. Conclusions

Analyzing a software system along with its past maintenance history can lead to indications of future maintenance work. Preventative maintenance can help hold down the high cost of maintenance. By determining where maintenance might occur and improving that code, future problems and costs can be minimized.

Values calculated using the regression equation are not meant to be an exact value for the amount of maintenance that will occur on a given procedure. Rather they are meant to give an indication of how much maintenance will occur. Analyzing a number of procedures creates a ranking of these procedures which can be used to determine on which preventative maintenance would be most useful. Future maintenance (and costs) could be lessened with an approach similar to the one presented in the last section.

No single metric seems to be able to determine the overall quality of a software system. For this reason the multiple regression approach presented here gives a good indication of a method for developing a predictor for a system. For an organization to use this approach, it must first collect a significant amount of error data (on maintenance of systems) and the corresponding metric values. Second, fit the error data to the metrics using multiple regression analysis. All organizations develop software using different tools and different environments. Developing the multiple regression equation for a specific environment requires data on programs that is specific to the environment in which it is developed, the application, and the language used by the software development organization. After developing and verifying the predictor equation for the data collected, it is ready to be used in more productive ways. This equation should be applied during coding, testing and maintenance. During coding, complicated procedures can be considered for redesign. At the end of the coding phase, procedures can be ranked as to likelihood of necessary maintenance. Preventative maintenance at this point will hold down future costs.

We were only able to evaluate one system in one language developed in one environment, it is obvious that substantially more research must be performed in this area to further validate our model. Bringing down the cost of software maintenance is necessary, and we feel that this model using several software quality factors is the direction for predicting software maintainability.

Bibliography

[1] Brooks, Jr., F.P., *The Mythical Man Month*, Reading, MA, Addison-Wesley Publishing Co., 1982.

[2] Ramamoorthy, C.V., Prakash, A., Tsai, W., Usuda, Y., "Software Engineering: Problems and Perspectives," *IEEE Computer*, October 1984.

[3] Benington, H.D., "Production of Large Computer Programs," *Proceedings of ONR Symposium on Advanced Programming Methods for Digital Computers*, June 1956.

[4] Boehm, B.W., Brown,J.R., Lipow, M., "Quantitative Evaluation of Software Quality," *Proceedings Second International Conference on Software Engineering*, 1976.

[5] Balzer, R., Cheatham, T.E., Green, C., "Software Technology in the 1990's: Using a New Paradigm," *IEEE Computer*, November 1983.

[6] Yau, S.S., Collofello, J.S., "Some Stability Measures for Software Maintenance," *IEEE Transactions on Software Engineering*, November 1980.

[7] Lientz, B.P., Swanson, E.B., Tompkins, G.E., "Characteristics of Application Software Maintenance," *Communications of the ACM*, June 1978.

[8] Curtis, B., Sheppard, S.B., Milliman, P., Borst, M.A., Love, T., "Measuring the Psychological Complexity of Software Maintenance Tasks with the Halstead and McCabe Metrics," *IEEE Transactions on Software Engineering*, March 1979.

[9] Basili, V.R., Perricone, B.T., "Software Errors and Complexity: An Empirical Investigation," *Communications of the ACM*, January 1984.

[10] Swanson, E.B., "The Dimension of Maintenance," *Proceedings of the 2nd International Conference on Software Engineering*, October 1976.

[11] Yau, S.S., Collofello, J.S., MacGregor, T., "Ripple Effect Analysis of Software Maintenance," *Proceedings of the IEEE Computer Science and Applications Conference*, 1978.

[12] Basili, V.R., Selby, R.W., Phillips, T., "Metric Analysis and Data Validation Across FORTRAN Projects," *IEEE Transactions on Software Engineering*, November 1983.

[13] Halstead, M., *Elements of Software Science*, New York, NY, Elsevier North Holland, Inc., 1977.

[14] Henry, S.M., Kafura, D.G., "Software Structure Metrics Based on Information Flow," *IEEE Transactions on Software Engineering*, September 1981.

[15] Kafura, D., Reddy, G.R., "The Use of Software Complexity Metrics in Software Maintenance," *IEEE Transactions on Software Engineering*, March 1987.

[16] Conte, S.D., Dunsmore, H. E., Shen, V. Y., *Software Engineering Metrics and Models*, Menlo Park, CA, The Benjamin/Cummings Publishing Company, Inc., 1986.

[17] McCabe, T., "A Complexity Measure," *IEEE Transactions on Software Engineering*, December 1976.

[18] Elshoff, J.L., "Measuring Commercial PL/1 Programs Using Halstead's Criteria." SIGPLAN Notices, May 1976, pp. 38 - 46.

[19] Curtis, B.; Sheppard, S.B. Milliman, P. "Third Time Charm: Stronger Predictors of Programmer Performance by Software Complexity," Fourth International Conference on Software Engineering, 1979, pp. 356 - 360.

[20] Albrecht, A.J., and Gaffney, J.E., "Software Function, Source Lines of Code and Development Effort Prediction: A Software Science Validation," IEEE Transactions on Software Engineering, November 1983, pp. 639 - 648.

[21] Basili, V.R.; Selloy, R.W.; Phillips, T. "Metric Analysis and Data Validation Across FORTRAN Projects," IEEE Transactions on Software Engineering, November 1983, pp. 652 - 663.

[22] Konstam, A.H. and Wood, D.E., "Software Science Applied to APL," IEEE Transactions on Software Engineering, October 1985, pp. 994 - 1000.

[23] Woodfield, S., *Enhanced Effort Estimation by Extending Basic Programming Models to Include Modularity Factors*, Ph.D. Dissertation, Purdue University, Computer Science Department, 1980.

[24] Kafura, D., Henry, S., "Software Quality Metrics Based on Interconnectivity," *Journal of Systems and Software*, Vol. 2, 1982.

[25] Henry, S.M., "A Technique for Hiding Proprietary Details While Providing Sufficient Information for Researchers," *Journal of Systems and Software*, January 1988.

[26] Henry, S.M., Kafura, D.G., Harris, K., "On the Relationship Among Three Software Metrics," *Performance Evaluation Review*, Vol. 10, No. 1, Spring 1981.

[27] Canning, J.T., *The Application of Software Metrics to Large-Scale Systems*, Ph.D. Dissertation, Virginia Tech, Computer Science Department, April 1985.

[28] Myers, R.H., *Classical and Modern Regression with Applications*, Boston, MA, Duxbury Press, 1987.

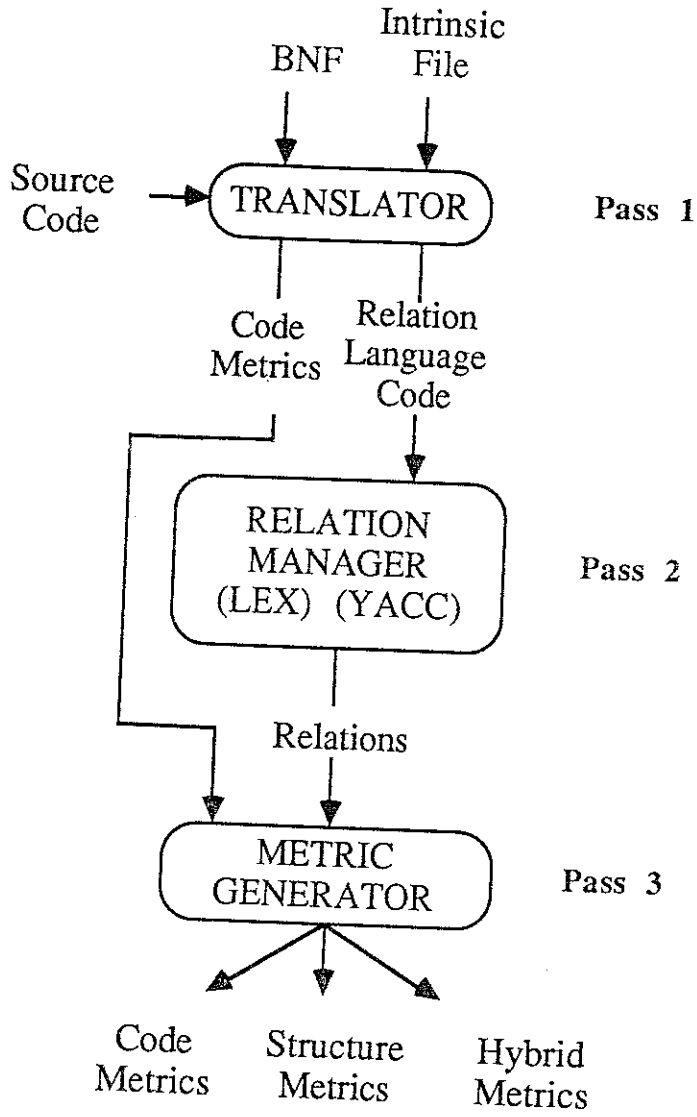

Figure 1. Software Metric Analyzer