# A Model Based on Software Quality Factors which Predicts Maintainability

Steven Wake and Sallie Henry

TR 88-8

# A Model Based on Software Quality Factors which Predicts Maintainability

by

Steve Wake
and
Sallie Henry

Computer Science Department
Virginia Tech
Blacksburg, VA 24061
(703) 961-2498

# A Model Based on Software Quality Factors which Predicts Maintainability


## Abstract

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 estimates that fifty percent of the development cost of a software system is for maintenance activities [BROF82]. 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 quality 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. We have developed a model which uses several software quality metrics as parameters to predict maintenance activity.

# I. Introduction

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 [RAMC84, YAUS80]. Lientz suggests that "maintenance and enhancement tend to be viewed by management as at least somewhat more important than new application software development" [LIEB78]. Curtis states that "more time is spent maintaining existing software than developing new code" [CURB79]. Since "the cost of correcting program errors can (and typically does) increase enormously with time to discovery," it is important to find these errors as early as possible [BASV84].

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 suggests that metrics can be used for maintenance purposes during the requirements, implementation, testing, and maintenance stages [RAMC84]. 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 have developed a software metric to measure the ripple effect of modifications in a software system [YAUS80, YAUS78]. 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 an impact on many other modules in the system. This leads to high maintenance costs and low system reliability.

Basili has attempted to determine the correspondence between the software science measures of Halstead 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 [BASV83, HALM77]. 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.

Henry and Kafura used the information flow metric to analyze the UNIX operating system [HENS81a]. They chose UNIX for several reasons, including its large size and the fact that it is not an experimental system but a system designed for users. They found that a high complexity shows stress points in the system or inadequate refinement of a procedure. A high correlation was found between changes in the system and the complexity of the procedures.

Kafura and Reddy studied the use of software complexity metrics on several versions of the same software system [KAFD87]. 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 fact 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."

Actual data must be used in order to perform a valid research effort using software quality metrics to predict system maintenance. Obtaining real data is a difficult task for academicians. If student data is used, the experiment may be conducted in too controlled an environment and not reflect "real world" conditions. Thus the best data comes from industry. Since most data from

industry comes from proprietary software, it is difficult to obtain. Pass 1 of the software metric analyzer described in Section II disguises the source code into an intermediate language so that the source code can be measured without revealing the data structures or actual algorithms [HENS88]. A research relationship between the software engineering group at Virginia Tech and a major software vendor has been ongoing for the last several years. The vendor understands the need of academicians to have access to real data. Cooperative research benefits both parties by improving the quality of their software by providing real data for software engineering researchers.

Section II describes the various software metrics and the software metric analyzer used in this study. In Section III, the software system which was measured and evaluated for maintenance activity is described. The models developed in this study which use software quality metrics to predict maintenance activity are presented in Section IV. Finally, Section V contains our conclusions and potential directions for future work.

## 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 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 [CONS86].

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 [MCCT76]. The cyclomatic complexity for a graph with e edges and n nodes is:

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

The third set of code metrics are Halstead's Software Science Indicators. 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 [HALM77]. 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 measurement 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 [HENS81a]. 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 [WOOS80]. 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 [KAFD82]. 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 [HENS88].

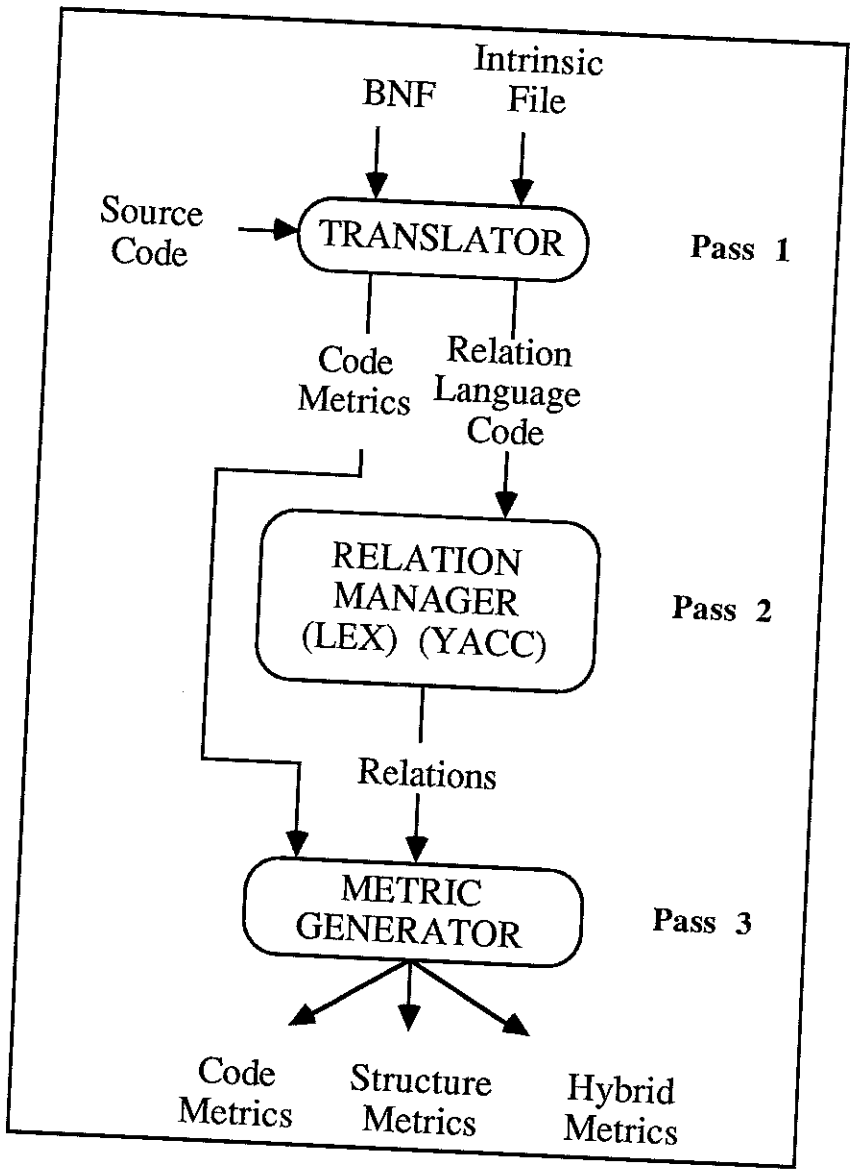


**Figure 1. Software Metric Analyzer**

## 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). 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.

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 latest 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 shows the interrelationships among the various metrics discussed previously and the number of lines of code changed in the procedures. A discussion of the multiple regression model and our statistical analysis is presented. 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 between the code metrics. This occurs because the code metrics are attempting to measure the same aspect of the code. However, there are low correlations between 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 [HENS81b, CANJ85].

| | 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 |

**Table 1. Intermetric Correlations**

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 during the maintenance phase of the software life cycle. Lines of code changed is the dependent variable 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 found that a single metric could not

adequately determine a good predictor. 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).

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 [MYER87].

At this point, we divide our data into two groups: three quarters of the data is used to develop a prediction model; the final quarter of the data is used to show that the model does predict maintainability by using the regression equation. This is an attempt to validate the model developed.

In order to pick the best multiple regression model possible, the statistics are developed for all possible models. This gives an indication of the best models although it does not necessarily give the "best" model since each of these statistics may choose a different model as best. The top models as chosen by each of the three statistics are shown in the following tables. As expected, these statistics do not agree as to the best model but choose a set of models that do consistently well throughout each statistic. These models and their associated statistic values are presented in Table 5.

| |
|---|
| 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 2. Top 5 Models Selected by PRESS Statistic**

| |
|---|
| 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 3. Top 5 Models Selected by MSE**

| |
|---|
| NLC = 1.47735619 + 0.000054638 E - 0.10017668 V(G) - 0.0000067303 WOOD |
| NLC = 1.53289519 + 0.000054777 E - 0.09957385 V(G) - 0.0000047908 WOOD - 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 4. Top 5 Models Selected by $C_p$**

$NLC = 0.42997221 + 0.000050156\ X_4 - 0.000000199210\ X_8$

$NLC = 0.45087158 + 0.000049895\ X_4 - 0.000173851\ X_7$

$NLC = 0.60631548 + 0.000050843\ X_4 - 0.000029819\ X_6 - 0.000000177341\ X_8$

$NLC = 0.33675906 + 0.000049889\ X_4$

$NLC = 1.51830192 + 0.000054724\ X_4 - 0.10084685\ X_5 - 0.000000161798\ X_8$

$NLC = 1.45518829 + 0.00005456\ X_4 - 0.10199539\ X_5$

**Table 5. Best Overall Candidate Models**

After choosing a set of best models, they can each be run against our unused data. By summing the squared error between the predicted value and the actual value an indication might be given if one of these models is better or worse than the others. Once again, it is observed that all of these models are predicting equally well.

## Prediction Example

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 equared error and be used to predict the amount of maintenance to occur on a given procedure. 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 3.31 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 0.0389 for NLC. This procedure required no modifications. A programmer would want to concentrate on the higher–valued procedure if there is time to do preventative maintenance.

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 an 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. These prediction equations should be applied and evaluated constantly during the coding, testing, and maintenance phases of the software life cycle. When a module or set of modules is predicted to require a large amount of maintenance, a redesign should be considered.

## V. Conclusions

No one metric can determine the overall quality of a system. For this reason, we believe that the use of the multiple regression model described in this paper is the best method to develop a predictor of system maintenance. The use of multiple metrics as parameters to our model presents the best overview of the system.

Although some of the metrics did not contribute to the model, that may be due to the development environment, the programming language or the system application. We are **not** saying that these metrics are not meaningful. In other environments, we would expect that the metric coefficients to the model could change drastically.

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 quality.

## Bibliography


[BASV84] Basili, V.R., Perricone, B.T., "Software Errors and Complexity: An Empirical Investigation," *Communications of the ACM*, January 1984.

[BASV83] Basili, V.R., Selby, R.W., Phillips, T., "Metric Analysis and Data Validation Across Fortran Projects," *IEEE Transactions on Software Engineering*, November 1983.

[BROF82] Brooks, Jr., F.P., *The Mythical Man-Month*, Reading, MA, Addison-Wesley Publishing Co., 1982.

[CANJ85] Canning, J.T., *The Application of Software Metrics to Large-Scale Systems*, Ph.D. Dissertation, Virginia Tech, Computer Science Department, April 1985.

[CONS86] Conte, S.D., Dunsmore, H. E., Shen, V. Y., *Software Engineering Metrics and Models*, Menlo Park, CA, The Benjamin/Cummings Publishing Company, Inc., 1986.

[CURB79] 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.

[HALM77] Halstead, M., *Elements of Software Science*, New York, NY, Elsevier North Holland, Inc., 1977.

[HENS81a] Henry, S.M., Kafura, D.G., "Software Structure Metrics Based on Information Flow," *IEEE Transactions on Software Engineering*, September 1981.

[HENS81b] Henry, S.M., Kafura, D.G., Harris, K., "On the Relationship Among Three Software Metrics," *Performance Evaluation Review*, Vol. 10, No. 1, Spring 1981.

[HENS88] Henry, S.M., "A Technique for Hiding Proprietary Details While Providing Sufficient Information for Researchers," *Journal of Systems and Software*, January 1988.

[KAFD82] Kafura, D., Henry, S., "Software Quality Metrics Based on Interconnectivity," *Journal of Systems and Software*, Vol. 2, 1982.

[KAFD87] Kafura, D., Reddy, G.R., "The Use of Software Complexity Metrics in Software Maintenance," *IEEE Transactions on Software Engineering*, March 1987.

[LIEB78] Lientz, B.P., Swanson, E.B., Tompkins, G.E., "Characteristics of Application Software Maintenance," *Communications of the ACM*, June 1978.

[MCCT76] McCabe, T., "A Complexity Measure," *IEEE Transactions on Software Engineering*, December 1976.

[MYER87] Myers, R.H., *Classical and Modern Regression with Applications*, Boston, MA, Duxbury Press, 1987.

[RAMC84] Ramamoorthy, C.V., Prakash, A., Tsai, W., Usuda, Y., "Software Engineering: Problems and Perspectives," *IEEE Computer*, October 1984.

[WOOS80] Woodfield, S., *Enhanced Effort Estimation by Extending Basic Programming Models to Include Modularity Factors*, Ph.D. Disertation, Purdue University, Computer Science Department, 1980.

[YAUS78] Yau, S.S., Collofello, J.S., MacGregor, T., "Ripple Effect Analysis of Software Maintenance," *Proceedings of the IEEE Computer Science and Applications Conference*, 1978.

[YAUS80] Yau, S.S., Collofello, J.S., "Some Stability Measures for Software Maintenance," *IEEE Transactions on Software Engineering*, November 1980.