The Use of Complexity Metrics Throughout the Software Lifecycle*

Sallie Henry, Steve Wake, and Wei Li

TR 92-59

Department of Computer Science
Virginia Polytechnic Institute and State University
Blacksburg, Virginia 24061

December 23, 1992

Reprinted from Proceedings of Interface 1990, The Twenty-Second Symposium on the Interface: Computer Science and Statistics.

The Use of Complexity Metrics Throughout the Software Lifecycle

by
Sallie Henry
Steve Wake
Wei Li

Computer Science Department
Virginia Tech
Blacksburg, VA 24061
(703) 231-7584
henry@vtodie.cs.vt.edu

ABSTRACT

Software metrics attempt to uncover difficult or complex components of a software system. The hypothesis is that complex components are more difficult to understand, hence they are hard to maintain and more prone to error. Discovery of these complex components can aid the software developer in 1) selection of which components to redesign, 2) direct the testing effort and 3) give an indication of the maintenance effort required.

Previous studies have demonstrated two main concepts. First, there exists a high correlation between design complexity and source code complexity. Secondly, metrics applied to source code have a high correlation to the maintenance activity needed.

The results of this previous research motivates us to develop a methodology which uses complexity metrics throughout the software life cycle. Programmer productivity may be increased and software development cost may be reduced if error prone software is discovered early in the life cycle.

I. INTRODUCTION

In the last decade, the field of computer science has undergone a revolution. It has started the move from a mysterious art form to a detailed science. The vehicle for this progress has been the rising popularity of the field of software engineering. This innovative area of computer science has brought about a number of changes in the way we think of (and work with) the development of software. Due to this renovation, a field that started with little or no design techniques, and unstructured, unreliable software has progressed to a point where a plethora of techniques exist to improve the quality of a program design as well as that of the resultant software. The popularity of structured design and coding techniques prove that there is widespread belief that the overall product produced using these ideas is somehow better. Statistics seem to indicate that this belief is true. Until recently, however, there existed no proven technique for quantitatively showing that one program is better than its functional equivalent. In the past few years, the use of software complexity metrics seems to indicate that such a comparison is not only possible, but also valid.

Several recent studies have shown that software metrics are good predictors of maintenance activities. These studies correlated number of errors found in the code and number of lines of source code changed with the metric values [CANJ85], [HENS90], [REDG84], [SELC88].

A typical software life cycle consists of requirements definitions, program design, implementation, testing, and finally, maintenance. The portion of the life cycle that is of interest to this research is the design and implementation with the inclusion of software metrics. Figure 1 contains one approach of this part of the software life cycle using complexity metrics.

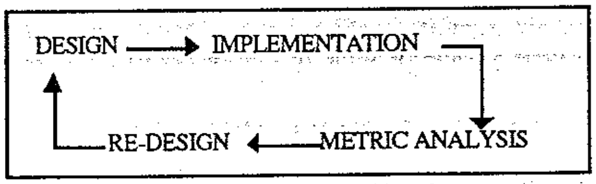

Figure 1. Diagram of Currently Used Software Life Cycle With Metrics

First, a design is created and implemented in software. At that point, software complexity metrics are generated for the source code. If necessary, as indicated by the metrics, the cycle returns to the design phase. Ideally, the software life cycle can be "reduced" to that in Figure 2, where the same metrics are generated during the design phase, before code implementation. This modified cycle eliminates the generation of undesirable source code, since it is possible to use the metrics, exactly as before, only earlier. The goal of this study is to indicate the plausibility of using the "reduced" cycle to increase the efficiency of the software development process, by implementing metric analysis as early as possible. In this research, PDL code is used to analyze designs [SELC88]. The PDL is Ada-like, however, any PDL with an imposed syntax could be used.

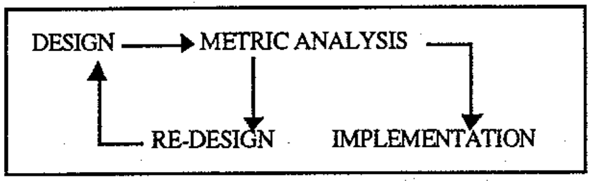

Figure 2. Diagram of Proposed Reduced Software Life Cycle With Metrics

The goal of shortening the loop in the life cycle is highly dependent on the ability to perform the analysis on the design, along with the need for evidence that the metric values produced from the design reflect the complexity and the maintainability of the resultant source code. To facilitate this ability, a software metric generator (which for purposes of this study may be considered a "black box") is provided that takes the design as input and produces a number of complexity metric values as output. For a more detailed explanation of the metric generator, see [HENS88].

Some of the existing metrics are qualitative and therefore non-automatable. These types of measures are not considered in this study. Here the focus is on metrics that are both quantitative and automatable. Metrics of this type can be put into two general categories: code metrics and structure metrics. In general, code metrics are those that measure an attribute such as length, number of control statements, number of tokens, etc. That is, code metrics produce a "count" of some feature of the source program. Structure metrics attempt to capture the interconnectivity of the components of the source program. Although both code and structure metrics result in a number that somehow represents the "goodness" of a program, it has been shown that the two types of metrics are measuring different features of the source systems [HENS81]. Note these metrics are static and do not consider the dynamics of the program.

Code Metrics

Many code metrics have been proposed in the recent past. An effort has been made to limit this discussion to a few of the more popular ones that are typical of this type of measure. They include lines of code, parts of Halstead's Software Science, and McCabe's Cyclomatic Complexity. Each of these metrics has been extensively validated [CANJ85] [ELSJ78] [REDG84].

Lines Of Code
The most familiar software measure is the count of the lines of code with a unit of *LOC*, or, for large programs, *KLOC* (thousands of lines of code). For this study, the definition used is the following: A line of code is counted as the line or lines between semicolons, where intrinsic semicolons are assumed at both the beginning and the end of the source file. This specifically includes all lines containing executable and non-executable statements, program headers, and declarations.

Halstead's Software Science
A natural weighting scheme used by Halstead in his family of metrics (commonly called Software Science indicators [HALM77]) is a count of the number of "tokens," which are units distinguishable by a compiler. All of Halstead's metrics are based on the following definitions:

n_1	=	the number of unique operands.
n_2	=	the number of unique operators.
n	=	$n_1 + n_2$
N_1	=	the total number of operands.
N_2	=	the total number of operators.
N	=	$N_1 + N_2$

Three of the software science metrics, N, V, and E, are used in this research. The metric N is simply a count of the total number of tokens expressed as the number of operands plus the number of operators.

V represents the number of bits required to store the program in memory.

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

The final Halstead metric examined is effort (*E*). The effort metric, which is used to indicate the effort of understanding, is dependent on the volume (*V*) and the difficulty (*D*). The unit of measurement of *E* is elementary mental discriminations which represents the difficulty of making the mental comparisons required to implement the algorithm.

$$E = V \times D, \text{ where } D = \frac{n_1}{2} \times \frac{N_2}{n_2}.$$

McCabe's Cyclomatic Complexity
McCabe's metric [MCCT76] is designed to indicate the testability and maintainability of a procedure by measuring the number of "linearly independent" paths through the program. To determine the paths, the procedure is represented as a strongly connected graph with one unique

entry and exit point. The nodes are sequential blocks of code, and the edges are decisions causing a branch. The complexity is given by:

$$V(G) = E - N + 2, \quad \text{where}$$
E = the number of edges in the graph
N = the number of nodes in the graph.

According to McCabe, $V(G) = 10$ is a reasonable upper limit for the complexity of a single component of a program. Throughout this paper, McCabe's Cyclomatic Complexity is often abbreviated as *CC*.

Structure Metric

It seems reasonable that a more complete measure will need to do more than simple counts of lines or tokens in order to fully capture the complexity of a module. This is due to the fact that within a program, there is a great deal of interaction among modules. Code metrics ignore these dependencies, implicitly assuming that each individual component of a program is a separate entity. Conversely, structure metrics attempt to quantify the module interactions using the assumption that the static inter-dependencies involved contribute to the overall complexity of the program units, and ultimately to that of the entire program. In this study, the structure metric examined is Henry and Kafura's Information Flow metric.

Henry and Kafura [HENS81] developed a metric based on the information flow connections between a procedure and its environment called "fan-in" and "fan-out" which are defined as:

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.

The complexity for a procedure is defined as:
$$C_p = (\mathit{fan\text{-}in} * \mathit{fan\text{-}out})^2.$$

The next section describes previous research studies. Section III presents a methodology for using metrics throughout the software life cycle. The statistics used to validate the methodology is discussed in section IV. Finally, our conclusions are given in section V.

II. BACKGROUND WORK

In this section, we examine several previously done studies in the area of software metrics. These studies are in the areas of design metrics, which predict the complexity of the code, and code metrics, which predict the amount of maintenance required.

Henry and Selig [HENS90b] analyzed PDL designs and the resulting Pascal code which were collected from the undergraduate software engineering course at Virginia Tech. The goal of the course is to expose the student to a real world environment of teams responsible for designing a system and "hiring" other class members to code the system. Each team would then integrate the system into the completed program. These systems range from 2000 to 8000 lines of Pascal code.

The PDL code and the Pascal code are run through a software metrics analyzer to calculate the values of various software metrics. The Pascal procedures were grouped together into modules to provide a one to one correspondence with the PDL modules. After elimination of outliers, there were 981 modules to analyze from 27 different projects.

Modules were grouped by level of refinement, where a highly refined module would contain code-like specifications, a low refinement module contains many natural language descriptions, and an average refinement module would be somewhere in between.

Each level of refinement was analyzed individually using correlations to determine the trends of the data and simple linear regression to develop predictor equations of the source code quality from the metric values of the design modules.

Another study in the area of measuring the complexity of software design was done by Henry and Goff [HENS89]. The purpose of the study was to determine if a graphical design language could be measured using established software quality metrics, and to show that these software quality metrics can be applied early in the life cycle to predict resultant code quality.

The graphical design language used is GPL, the graphical programming language of the Dialogue Management System developed at Virginia Tech [HARH85]. An approach for translating graphical designs into relation language, a language used in the software metric analyzer, was developed in order to generate a consistent set of metrics for both graphical and textual designs.

Analysis was done between the design and the resultant source code to show that there were high correlations between the code metrics in GPL designs. A simple linear regression was performed between the complexity of the design, the independent variable, and the complexity of the source code, the dependent variable. This regression produced predictor equations, which did a good job of predicting the source code quality from the design quality.

Wake and Henry [WAKS88] measured a 15,000 line C system consisting of approximately 193 procedures. This system was a "real world" system obtained from a major software vendor. Error data, consisting of modifications to the system after release for bug or performance fixes, was also obtained.

The number of times a function had error fixes done to it along with the total number of lines changed to effect a fix was calculated for each function. These numbers give an indication of the amount of maintenance which occurred with this system.

The system was measured using the software metric analyzer, in order to obtain metric values for each function in the system. Intermetric correlations were performed to see which metrics seemed to be measuring the same aspects of the code. Code metrics and structure metrics correlated well with metrics of the same type, but poorly with metrics of other types.

Performing correlations between the error data and the individual metrics would not give satisfactory results, since we do not feel that any one metric will perform equally well in any environment. Therefore, we tried to find a process that would determine the best set of metrics for the given environment. To this end, we used the multiple regression model to find the metric or metrics that best predicted the amount of maintenance necessary.

Henry and Lewis [HENS90a] conducted an experiment to integrate metric use into a commercial budget driven environment. Over 7000 procedures form a specific internal release of a large product written in an in-house, high-level language similar to Ada.

Tolerances were developed for the values of the various metrics which were measured on the software system. Two levels of concern were developed, a flag to indicate that the specific metric values for a procedure need to be justified in some manner, and an alarm which shows that values are too high.

An error history was used to validate the metrics used in the analysis. Correlations were done between the metric values and the errors from the error history. Regression analysis was also performed between multiple metrics and the number of errors to develop a predictor equation.

Based on these studies and the works of several other metric researchers, we feel that the promising results can guide us in developing a metrics methodology which can be incorporated into the software development life cycle. This research is explained in the next section.

III. LIFE CYCLE METHODOLOGY

Henry and Selig's study [HENS90b] shows the possibility of measuring the design with the existing metrics. Henry and Wakes' study [WAKS88] shows the possibility of predicting the maintainability using the metrics collected from the code. The study by Henry and Lewis [HENS90a] suggests a methodology which incorporates software metrics throughout the software life cycle. All of these previous results motivate us to conduct an experiment of applying software metrics throughout the software life cycle.

Several factors are considered in designing the experiment. The first factor is what metrics to use in the experiment. Among all the available software quality metrics, only those which are quantitative and automatable are used in this experiment. For more detailed information concerning the metrics, refer to Section I.

The second factor is when to collect these metrics. Metrics collected from the code have been very successful in indicating error prone modules, uncovering difficult modules, and predicting the maintenance work required. But metrics collected from code are considered too late to have significant influence on the redesign effort and the reduction of the overall budget. Collecting metrics earlier in the life cycle guides the redesign effort and improves the quality of the end product which reduces the overall budget. In this experiment, metrics are collected from both the design and the code.

The third factor is the validation of the metrics. This depends on how the metric values are to be interpreted and used. In this research, the goal is to use the metrics to predict maintainability and guide the redesign effort, therefore, the successful validation of the metrics depends on how much of the error variance is accounted for by the metrics. Therefore, the error history data are collected. The metrics collected from the design are to be used to predict the maintainability of the software product. With the metrics collected from the code for the same design, the results from previous studies will be verified.

The use of the statistical model and the interpretation of the statistical analysis are the keys to the success development of the methodology. Wake and Henry [WAKS88] indicated in their study that performing correlations between the error data and the individual metrics would not give satisfactory results since they do not feel that any one metric will perform equally well in any environment. Therefore, to determine the best set of metrics for the given environment, a regression model would give us more information about the relationship of the error data and the design metrics. Several studies conducted by Henry and Selig, Wake, and Goff [HENS90b] [WAKS88] [HENS89] have found that metrics in different categories measure different aspects of the software, therefore, metrics from different categories together should predict maintainability better than any single category. The multiple linear regression has been successfully used in the study conducted by Henry and Wake [WAKS88] to predict maintainability from metrics collected from code. In this experiment, the multiple linear regression is used to predict maintainability from metrics collected from design.

Since the successful development of the methodology depends on the significant result from predicting maintainability using design metrics, multiple linear regression is a natural choice as a statistical model in the experiment. In deciding the independent variables (metrics) that should be included in the model, the result from Henry and Selig [HENS90b] study is considered. In their study, an inter-metric correlation is performed. They found that there is a high correlation among code metrics, and a low correlation between code metrics and design metrics. This makes us think that different types of metrics measure different aspects of the software. That means multicollinearity exists among code metrics. Therefore, different types of metrics should be included in the model, and the number of highly correlated metrics should be limited.

IV. STATISTICAL MODEL

This section describes the statistical model used in the study.

Statistical Model

Multiple linear regression is chosen as the experiment statistical model. The different metric values are used as independent variables in the regression, the number of errors as the dependent variable. Multiple linear regression will be performed at the procedure and system level. The multiple linear regression model for the experiment is defined as :

$$Y = b_0 + b_1*X_1 + b_2*X_2 + b_3*X_3 + b_4*X_4 + b_5*X_5 + b_6*X_6 + b_7*X_7 + e,$$

where Y is the dependent variable representing the number of errors. X1 through X7 are independent variables representing nine different metric values. The e is the error term in the regression. The nine independent variables are defined as follows:

X_1 :	Lines of Code.
X_2 :	Halstead N.
X_3 :	Halstead E.
X_4 :	Halstead V.
X_5 :	McCabe's Cyclomatic Complexity.
X_6 :	Henry-Kafura Information Flow Complexity.
X_7 :	Woodfield's Review Complexity.

Statistical Analysis

The statistical analysis has five steps.

Step 1: Dividing the data
Randomly divide the data into two equal groups, say group A and group B. Group A is used in step 2 and step 3. Dividing data into two groups can avoid 'overfitting' problem in regression. Whenever variable selection routine is used, there is a possibility that the predicting equation is tailored just to fit the sample data. To avoid this problem, the data is divided into two groups. Only one group is used to perform the regression first. When the regression turns out to be significant. Another group of data is used to verify the equation. And finally, the whole set of data is used to perform the multiple linear regression again to get the best predicting equation.

Step 2: Significant test on the full model.
Using the F-test at 5% level to see if any prediction from the model is possible. The null hypothesis used is : H_0: $b_1=b_2=b_3=b_4=b_5=b_6=b_7=0$. The goal is to reject the null hypothesis which means that some prediction is possible.

Step 3: Variable Selection
If from step 2 some prediction is possible, we would like to see if a subset of independent variables could be picked which does essentially as good of a job as the whole set of independent variables do. Based on the previous studies, some metrics correlate very high with some other metrics. Since multicollinearity may exist among code metrics, the partial sum of squares of any code metrics given the rest of the code metrics may be near zero. This indicates that there might be some redundant predictors in the model. Therefore, not all of the code metrics are useful in predicting the error. A subset of all the metrics will be decided in the final prediction equation. There are three different variable selection procedures. They are forward, backward, and stepwise selection. There are four statistics that can be used to examine the quality of the prediction. They are R^2, adjusted-R^2, MSE (mean square error), and C(p). R^2 is the amount of total variability accounted for by the regression in the sample. R^2 increases when adding more predictors into the model. The adjusted-R^2 is the estimated amount of total variability accounted for by the regression in the population. It is adjusted for the number of predictors in the regression model so that it may or may not increase as more predictor are included in the model. The adjusted-R^2 is always less than R^2. They are both within the range of 0 and 1. The MSE is the measurement of the error variance in the regression. It does not give more information than the adjusted-R^2. It is possible that the regression model is biased (underfit) due to too few predictors in the model, or overfit due to redundant predictors. The quality predictor takes into account both possibilities. A small C(p) value

should indicate that the regression model does not have a serious underfit or overfit problem. In this experiment, adjusted-R^2, and C(p) are considered as the quality judgement for the regression equation.

Step 4: Cross Validation.
The cross validation procedure is used to determine if the prediction equation is tailored too much for the sample data used in performing the multiple linear regression. Any variable selection routine is using the sample to help determine the model. Thus, it is entirely possible that one will tailor the model too precisely to the data at hand. In this event the model will look good for the sample, but may not work at all in the population. To protect against this possibility, cross-validation should be used any time a variable selection routine is employed. For each data point in group B which is not involved in the regression, get the predicted Y value using the equation. Then test the correlation between the observed Y and the predicted Y at the 5% significance level. Also, the MSE_B and MSE_A are compared to see if the former is significantly larger than the latter. If that is the case, then the predicting equation is tailored for the sample data. If the correlation test turns out to be non-significant or the MSE_B is significantly larger than MSE_A, then the cross validation fails. If the cross validation is unsuccessful, the model has been tailored too precisely to the sample, so another try will be attempted with fewer predictors.

Step 5: Best prediction equation
If the cross-validation is successful, the terminal model is run again using all the data (group A and group B) to obtain the most stable estimates of the regression coefficients.

Development of the methodology

The traditional waterfall model of software development provides a systematic method to separate the development process into different stages with explicit communication boundary between two consequent stages. This methodology proposes that each phase be subjected to verification and validation to ensure that the implementation satisfies the specification. Most of the verification and validation after each phase finishes are 1) to check if the documents are complete; 2) to see if a certain discipline is enforced; 3) to check if the documents produced are consistent with those accepted at the beginning of the phase. The new methodology provides 1) the quantitative evaluation of the design; 2) feedback information to help redesign effort; 3) improvement in software development efficiency; 4) reduction of the budget.

The methodology suggested in this research incorporates all of those which are not provided by the waterfall model.

V. CONCLUSIONS

Upon the completion of the experiment, the following questions may be answered:

1. Can the design be measured?
2. Are the design metrics meaningful in predicting the resultant code quality?
3. Are the design metrics meaningful in predicting the maintainability of the software product?
4. Are there threshold values in design metrics that could be used as the redesign criteria?
5. Can the design metrics be used to guide test effort?
6. Can the design metrics be used to assist the project manager in management of the project ?

BIBLIOGRAPHY

[CANJ85] Canning, J.T., The Application of Software Metrics to Large-Scale Systems, Ph.D. Thesis, Computer Science Department, Virginia Polytechnic Institute, Blacksburg, Virginia, April 1985.

[ELSJ78] Elshoff, J.L., "An Investigation Into The Effect of The Counting Method Used on Software Science Measurements," *ACM SIGPLAN Notices*, Vol. 13, No. 2, pp. 30-45, February 1978.

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

[HARH85] Hartson, H.R., *Advances in Human-Computer Interaction*, Ablex Publishing Company, Norwood, NJ, 1985.

[HENS81] Henry, S.M. and Kafura, D.G. "Software Structure Metrics Based on Information Flow," *IEEE Transactions on Software Engineering*, (7,5), September, 1981, pp. 510 - 518.

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

[HENS89] Henry, S.M. and Goff, R.A., "Complexity Measurement of a Graphical Programming Language," *Software Practice and Experience* (November 1989).

[HENS90a] Henry, S.M. and Lewis, J.A., "Integrating Metrics into a Large-Scale Software Development Environment," *Journal of Systems and Software* special issue on Using Software Metrics (to appear 1990).

[HENS90b] Henry, S.M., and Selig, C.L., "Design Metrics which Predict Source Code Quality," *IEEE Software* special issue on development metrics, (7,2), March, 1990, pp. 36-44.

[MCCT76] McCabe, T.J., "A Complexity Measure," *IEEE Transactions on Software Engineering*, Vol. SE-2, No. 4, pp. 308-320, December 1976.

[REDG84] Reddy, G., *Analysis of a DataBase Management System Using Software Metrics*, M.S. Thesis, Computer Science Department, Virginia Polytechnic Institute, Blacksburg, Virginia, June 1984.

[SELC88] Selig, C.A. and Henry, S.M., "A Design Tool Used to Quantitatively Evaluate Student Projects," *Proceedings of the 1988 ACM SIGCSE Conference*, Atlanta, Georgia, February 1988.

[WAKS88] Wake, S.A., and Henry, S.M., "A Model Based on Software Quality Factors which Predicts Maintainability," *Proceedings of the 1988 IEEE Conference of Software Maintenance*, October, 1988, pp. 382 - 389.