A Metric Tool for Predicting Source Code Quality from a PDL Design

Sallie Henry
Calvin Selig

TR 87-23

A Metric Tool

for

Predicting Source Code Quality

from a

PDL Design

by

Sallie Henry
and
Calvin Selig

VIRGINIA TECH
Computer Science Department
Blacksburg, VA 24061

Abstract

The software crisis has increased the demand for automated tools to assist software developers in the production of quality software. Quality metrics have given software developers a tool to measure software quality. These measurements, however, are available only after the software has been produced. Due to high cost, software managers are reluctant, to redesign and reimplement low quality software. Ideally, a life cycle which allows early measurement of software quality is a necessary ingredient to solving the software crisis. This paper describes an automated tool for predicting software quality at design time.

Keywords

Software Quality Metrics, Program Design Languages (PDLs), Design Metrics, Automated Metric Tools, UNIX, Henry and Kafura's Information Flow Metric, Halstead's Software Science, McCabe's Cyclomatic Complexity.

I. Introduction

In general, the software life cycle consists of requirements definition, program design, implementation, testing, and finally, maintenance. The portion of the cycle that is of interest to this research is that of design and implementation with the inclusion of software quality metrics. Figure 1 contains a diagram of this part of the software life cycle using complexity metrics. First, a design is created and implemented in software. At that point, software quality 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 metrics are generated during the design phase, before code implementation. This modified cycle will eliminate 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.

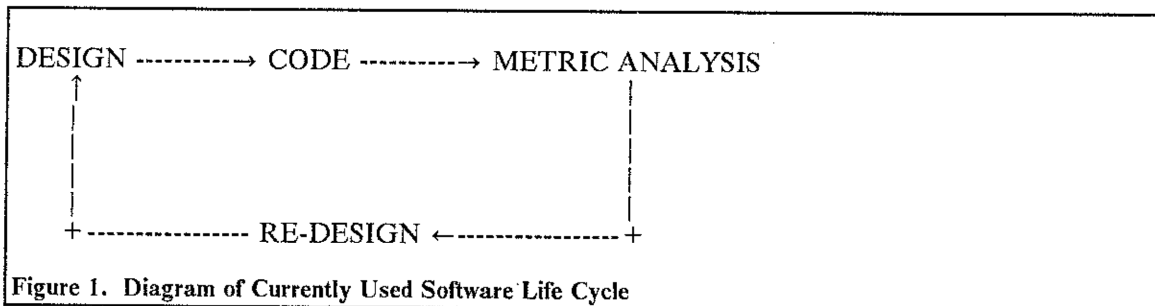

Figure 1. Diagram of Currently Used Software Life Cycle

The goal of shortening the loop in the life cycle is highly dependent on the ability to perform the metrical measures on the design, along with the need for evidence that the metric values produced from the design reflect the quality of the resultant source code. To facilitate this ability, a software metric analyzer is provided that takes as input either the design or the source code and produces, as output, a number of complexity metric values.

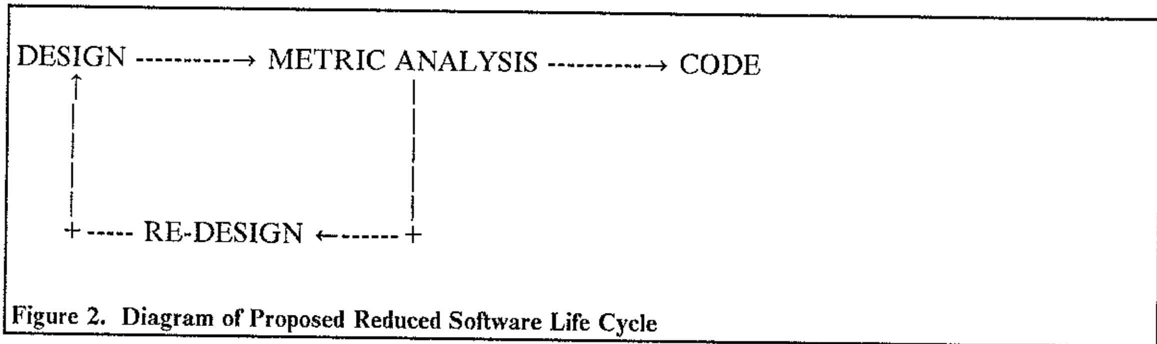

Figure 2. Diagram of Proposed Reduced Software Life Cycle

The metric analyzer requires syntactically correct code. When using the analyzer at design time, input consists of syntactically correct program design language (PDL) code. To this end, a PDL was developed for use in this research. This design language has some Pascal-like and ADA-like constructs, but it is sufficient to say this tool is a general PDL. For a complete definition of the PDL, the interested reader is referred to [SELC 87].

The next section describes in detail the software metric analyzer. Exact definition of the metrics produced by the tool are not provided but left to the reader. In section III, a brief discussion of the data collection and preparation is given. The data analysis and resultant prediction equations for source code quality are given in section IV. In addition, section IV also contains the inter-metric correlations. Finally, section V contains our conclusion.

II. Description of Software Metric Analyzer

A software quality metric analyzer, which takes as input PDL or source code and produces several software metrics, has been developed for use in our research. A general relation language has been successfully used as a tool to express the intermediate form of the design or source code [HENS88]. This intermediate form is then translated into a set of relations which are interpreted to produce metrics. The software quality metric analyzer produces three code metrics, three structure metrics, and several hybrid metrics. This analyzer is based on **LEX** (a lexical analyzer) and **YACC** (Yet Another Compiler Compiler) which are tools available with a **UNIX** environment. Hence, the analyzer requires a **UNIX** system.

The remainder of this section describes the details of the implementation of the software quality metric analyzer. For purposes of discussion, the analyzer is divided into distinct three passes. See Figure 3 for a diagram of the analyzer.

Pass 1

Pass one has as input the Backus-Naur form (BNF) grammar for the PDL or source language to be analyzed, the semantic routines which dictate processing for each production in the grammar, and the design or source code to be analyzed. A file containing the intrinsic (built-in) functions, peculiar to the source language is also input. For obvious reasons, these functions should not be treated as real functions; they actually act similar to complicated operators and as such are treated as operators. The source code to be analyzed is assumed to be syntactically correct.

Two files are output from pass one. The first file contains the language dependent metrics for each procedure: lines of code (LOC) [CONS 86], McCabe's Cyclomatic Complexity (CC) [MCCT 76],

and Halstead's Software Science indicators length, volume, and effort (N, V, and E respectively) [HALM 77]. These metrics are produced in pass one since this is the only pass which has the actual code necessary to generate them. The second file output from pass one contains the Relation Language code which is equivalent to the source code. Pass one is the only language-dependent portion of the analyzer. Current source languages processed are the PDL used in this study, Pascal, 'C', FORTRAN, and THLL, a language used by the United States Navy.

Pass 2

Pass two uses the **UNIX** tools **LEX** and **YACC**. The Relation Language code from pass one is translated into a "set of relations" [KAFD 82]. This set of relations is completely independent of the original language. Code can be processed one procedure at a time. An advantage is that the Relation Language code for the procedure is the only information necessary to generate its relations. An additional advantage is that source code could be translated into Relation Language code and then analyzed at a separate facility. This feature allows any proprietary details in the original source code to be hidden from the analysis process [HENS 88].

Pass 3

Three general classes of software metrics can be distinguished: *structure metrics* which are measures based on automated analysis of the system's design structure, *code metrics* which are measures based on implementation details, and *hybrid metrics*, which combine features off both structure and code metrics. As previously proposed by [CANJ 85], [HENS 81b], and [HENS 84], this research shows that the structure metrics are global indicators of software quality which can be taken early

in the life cycle, while code and hybrid metrics can be brought into use as more implementation details become visible.

Pass three and the associated implementations of the structure metrics are written in standard Pascal. The relations file from pass two generates the three structure metrics: Henry and Kafura's Information Flow metric [HENS 81a], McClure's Invocation metric [MCCC 78], and Woodfield's Review Complexity metric [WOOS 80]. Only the Information Flow metric was available for this study. The structure metrics and the code metrics (file one from pass one) produce the hybrid metrics.

A quantitative measurement of design structure can be defined only in terms of those features of the software product which have emerged during the (high-level) design phase. To define a numerical measure, structure metrics use only these features, components, and relationships among components. Note that the actual source code is not necessary to observe the interconnections among components of a system.

A structure measure based on the data relationships among components is the **Information Flow Metric** [HENS 79]. This metric identifies the sources (fan-in) and destinations (fan-out) of all data related to a given component. The data transmission may be through global data structures, parameters, or side-effects. The fan-in and fan-out are then used to compute a worst-case estimate of the communication "complexity" of this component. This complexity measure attempts to gauge the strength of the component's communication relationships with other components.

As previously stated, pass three is written completely in standard Pascal and is independent of a **UNIX** environment. The user is in complete control of the selection of the above metrics to be run and the method of viewing the metrics. The user decides which of the structure metrics he desires to apply to his system. In addition to running the structure metrics and examining them, the user

is allowed to define modules (a related collection of procedures) or levels (a related collection of modules). It is assumed that the user would like to view all related procedures as a single module, and likewise, view all related modules as a single level. This feature is especially useful for very large systems. Hardcopies of all reports are available at any time.

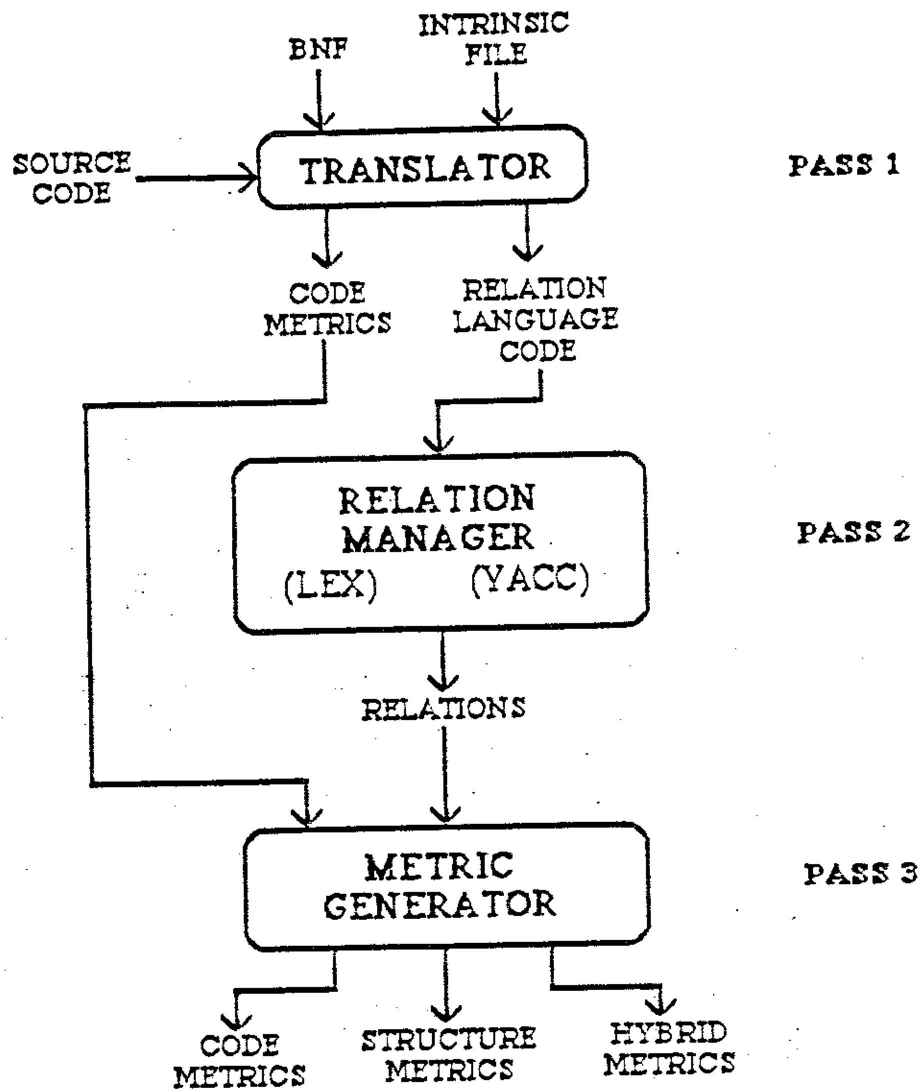

Figure 3. Diagram of the Software Metric Analyzer

III. Data Collection and Preparation

Research, such as that done for this study, is highly dependent on the availability of relevant data to analyze. In general, programs produced by students and used as data are not very well accepted. Due to the problem with obtaining "real world" data, the data used for this study is student generated, but there is a major difference between the data used here and that generally produced in academia. The programming is done in an environment that is as realistic as possible. Therefore, it is felt that even though the programs used are produced within an academic environment, they are substantially more realistic than that generally obtained under these conditions.

Over the past six years, PDL designs and the resultant Pascal source code has been collected from undergraduate senior-level software engineering courses at both Virginia Tech and the University of Wisconsin — LaCrosse. The project-oriented class is designed to teach students the basics of software engineering [HENS 83].

The goal of the project is to expose students to the experience of non-trivial program development in a "real world" environment where designing and implementing a project is not a single-person task. To this end, the class is divided into teams of three people. Each team is responsible for designing a system that upon completion will be from three thousand to five thousand lines of source code. This design includes hierarchy charts and module specifications written in the PDL. After a team has completed the specifications for their project, they "hire" classmates to implement the modules in Pascal. Finally, the design team must integrate the modules into the completed program.

The completed projects that have been furnished by the classes are varied. They range from two thousand to eight thousand lines of code, indicating the students' inability to accurately predict program size. Program function is also widespread. In an attempt to ensure that the completed designs and source are usable as data, students are given minimal design requirements. It was im-

possible to monitor all of the designs of all of the students; in fact, the students alone decided on "English-like" specifications, "code-like" specifications, or somewhere in-between.

After the PDL and Pascal code have been processed by the analyzer and metrics have been generated, procedures must be combined into modules. This is necessary because there must be a one-to-one correspondance between the design and source. The method used is simple. A single PDL procedure may be a definition of the function performed by several Pascal procedures. In order to equate the design and source, it is necessary to add the complexities of each of the Pascal routines. Combining procedures into modules is easily accomplished by using the module definition feature of the analyzer. In this way, the design complexity is directly comparable to the source complexity on a functional level.

IV. Data Analysis

It is desirable to examine the data as a whole in order to develop predictors for each metric. Preliminary results indicate that the code metrics are strongly dependent on the level of the refinement while the structure metrics are independent of refinement level [SELC 87]. A highly refined module would contain code-like specifications, while a low refinement level indicates the predominance of natural language specifications. To determine the validity of this hypothesis, the routines are divided into three categories; low, average, and high levels of refinement. Each level is analyzed individually, where the analysis consists of (1) correlations to determine the overall trends of the data, and (2) simple linear regression analysis to obtain the predictors. In addition to the prediction of the source quality, it would be convenient if an estimate of the amount of error involved in the prediction could be calculated. To this end, the extra statistics necessary to calculate a 95% confidence interval for a predicted source complexity are determined.

As mentioned above, it is informative to examine the metric values based on the level of refinement of the specifications. In order to accomplish this purpose, it is necessary to determine the refinement level for each routine to be examined. It seems reasonable that as refinement level increases, the length of the procedure will also increase, but more importantly, the number of control structures will be greater as well. Also, it appears necessary to consider the relative or normalized complexity values as opposed to examining the raw data, since, for example, the semantic difference between 1 and 5 is greater than that between 40 and 50. Using these ideas, an algorithm to automatically determine the completeness of the design has been created. It is interesting to note that the algorithm is completely dependent on code metric values, yet, as will be demonstrated, the Information Flow metric performs better than the code metrics. This again indicates that structure metrics are probably independent of refinement level.

The algorithm proceedes as follows: first a normalized value is calculated for McCabe's Cyclomatic Complexity and for Halstead's N metric for each procedure. In the next step of the algorithm the

level of refinement is determined using the normalized values. If the normalized McCabe value is 0.0, then it cannot be used to determine level, and the metric N is the sole determiner. Otherwise, the McCabe value is checked first, and if it appears fully refined, the level is either High or Middle, depending on N. If the McCabe value is of Middle refinement, the level is either Middle or Low, again depending on the value of the normalized N. Finally, if the McCabe value indicates a low level of refinement, the procedure is determined to be Low. The last step in the algorithm performs the actual regression at each level of refinement.

In order to determine the validity of the results of the algorithm, seven projects were examined and procedural refinement determined by hand. Comparison of the results of the hand-generated levels and those determined by the algorithm indicate that this method of setting a level of refinement was extremely accurate, being correct more than 99% of the time. Due to the strong results of the algorithm, it was used to generate the refinement level of all of the procedures. The results of the algorithm indicate that of the 981 modules in the study; 422 have a High level of refinement, 283 have a Middle level of refinement, and 276 have a Low level of refinement.

Table 1. Correlations by Level of Refinement

Low Level						
	LOC	**N**	**V**	**E**	**CC**	**INFO**
	0.610	0.512	0.552	0.211	0.405	0.903

Middle Level						
	LOC	**N**	**V**	**E**	**CC**	**INFO**
	0.701	0.731	0.756	0.577	0.867	0.904

High Level						
	LOC	**N**	**V**	**E**	**CC**	**INFO**
	0.810	0.830	0.807	0.706	0.902	0.792

The analysis results, shown in Table 1, show that the above expectations are generally true. In the tables, Henry and Kafura's Information Flow metric is abbreviated INFO. The Information Flow metric performed consistently well, with its lowest correlation value being 0.792. This result shows explicitly that the metric is independent of refinement level. The code metrics reacted as predicted. They do not perform well at a low level of refinement, and their correlations increase as level of refinement becomes greater. An interesting result is that the metrics perform quite well at even a middle refinement level. This indicates that after a minimum amount of detail is included in the specifications, the code metrics become useful measures.

Tables 2, 4, and 6 show the regression lines for each of the metrics at a Low, Middle, and High level of refinement respectively. As discussed above, the information in these tables may be used to form a prediction equation, given that the level of refinement can be determined, with a 95% degree of confidence. Tables 3, 5, and 7 give the extra statistics needed to predict a confidence interval for a specific design complexity value and refinement level using the formula:

$$Slope \times X + Intercept \pm 2 \times \sqrt{MSE \times (1/n + (X - X_{Mean})/SS_x)}$$ where

X is the independent variable or design complexity

Slope is the slope for the regression line

Intercept is the intercept for the regression line

MSE is the Mean Squared Error from the model

n is the total number of data points employed in the regression

X_{Mean} is the average value of the design complexities

SS_X is given by :

$$SS_X = \sum_{i=1}^{n} (X_i - X_{Mean})^2$$

[WALR 78]

Using the above equation and the prediction line for LOC as given above, the 95% confidence interval for a design complexity of 100, and a low level of refinement is given by:

$$1.126 \times 100 + 29.224 \pm 2 \times \sqrt{1.785E + 5 \times (1/276 + (100 - 11.388)/1.409E + 5)}$$

$$141.824 \pm 55.100$$

Another words, the actual value associated with the complexity 100 for the metric LOC will be in the range 86.724 — 196.924 95% of the time.

Table 2. Low Refinement Regression Line Equations and Statistics

Regression Line Information				
		Coef	Std Err	t-Value
LOC	Intercept Slope	29.224 1.126	2.236 0.088	13.069 12.737
N	Intercept Slope	226.137 1.410	18.032 0.143	12.541 9.877
V	Intercept Slope	1235.605 1.396	104.910 0.127	11.778 10.961
E	Intercept Slope	90760.893 2.570	16309.020 0.719	5.565 3.573
CC	Intercept Slope	3.630 6.117	1.286 0.833	2.823 7.341
INFO	Intercept Slope	81740.175 10.022	105225.480 0.286	0.777 35.106

Table 3. Extra Statistics for Low Refinement Level

Metric	*Statistics*			
	n	*MSE*	SS_x	X_{Mean}
LOC	276	1.785E+05	1.409E+05	11.388
N	276	7.024E+06	3.531E+06	56.127
V	276	3.227E+08	1.657E+08	280.500
E	276	8.709E+11	1.318E+11	6031.786
CC	276	7.690E+03	2.055E+02	1.279
INFO	276	3.710E+15	3.693E+13	45085.200

Table 4. Middle Refinement Regression Line Equations and Statistics

Regression Line Information				
		Coef	Std Err	t-Value
LOC	Intercept Slope	4.900 1.242	1.241 0.075	3.948 16.498
N	Intercept Slope	49.247 1.440	8.553 0.080	5.758 17.937
V	Intercept Slope	285.608 1.408	43.100 0.073	6.627 19.356
E	Intercept Slope	24147.773 1.459	4525.805 0.123	5.336 11.851
CC	Intercept Slope	0.789 1.932	0.259 0.066	3.045 29.219
INFO	Intercept Slope	-7258.697 5.231	14693.092 0.148	-0.494 35.447

Table 5. Extra Statistics for Middle Refinement Level

Metric	*Statistics*			
	n	*MSE*	SS_x	X_{Mean}
LOC	283	5.893E+04	3.820E+04	11.696
N	283	2.757E+06	1.812E+06	70.360
V	283	1.285E+08	6.481E+07	349.226
E	283	7.614E+11	3.576E+11	9348.640
CC	283	9.615E+03	2.577E+03	2.502
INFO	283	7.530E+13	2.752E+12	13747.350

This section has presented the statistical analysis for the data based on levels of refinement. It shows that the structure and hybrid metrics perform much better than the code metrics, and more importantly, proves that structure metrics are independent of level of refinement. With a minimal amount of information (the calling structure and parameters) the Information Flow metric can predict the quality of the resultant software. This facilitates a shorter life cycle for the production of quality software. Regression lines allow the calculation of prediction equations for the resultant source code complexity values to be predicted based on the design.

Table 6. High Refinement Regression Line Equations and Statistics

Regression Line Information				
		Coef	Std Err	t-Value
LOC	Intercept Slope	1.284 0.810	0.547 0.029	2.348 28.304
N	Intercept Slope	20.210 0.842	4.262 0.028	4.741 30.488
V	Intercept Slope	110.247 0.830	25.482 0.030	4.327 28.002
E	Intercept Slope	9522.419 0.756	3647.289 0.037	2.611 20.458
CC	Intercept Slope	0.644 0.791	0.133 0.018	4.829 42.840
INFO	Intercept Slope	-297874.630 40.602	958329.640 1.527	-0.311 26.582

Table 7. Extra Statistics for High Refinement Level

Metric	*Statistics*			
	n	*MSE*	SS_x	X_{Mean}
LOC	422	4.404E+04	6.717E+04	14.358
N	422	3.959E+06	5.590E+06	102.960
V	422	1.345E+08	1.954E+08	525.775
E	422	2.239E+12	3.916E+12	21439.980
CC	422	8.218E+03	1.313E+04	4.583
INFO	422	2.719E+17	1.650E+14	52433.400

Inter-metric Correlations

Table 8 contains the inter-metric correlations for the Pascal code. This information serves to verify studies previously done by [HENS 81b], [CANJ 85], [BASV 83], [REDG 84], and [LIH 87]. The table indicates that between code metrics there is, in general, a high degree of correlation. When comparing the code metrics with the structure metric (Information Flow), however, it is seen that

the results are quite close to zero. This result indicates that the code and structure metrics are measuring different aspects of the source code.

The inter-metric correlations, with respect to the design, show the same relationships as those using Pascal, and are given in Table 9. This is a desirable result since it indicates a consistency of measurement when comparing the designs to the resultant source code. It also lends credence to the usefulness of performing complexity measures at design time.

Table 8. Inter-metric Correlations for the Pascal code

Pascal Source Code						
Metric	**LOC**	**N**	**V**	**E**	**CC**	**INFO**
LOC						
N	0.893					
V	0.885	0.989				
E	0.521	0.749	0.711			
CC	0.629	0.776	0.781	0.492		
INFO	0.044	0.029	0.036	0.005	0.019	

Table 9. Inter-metric Correlations for PDL code

PDL Design Code						
Metric	**LOC**	**N**	**V**	**E**	**CC**	**INFO**
LOC						
N	0.894					
V	0.894	0.988				
E	0.465	0.725	0.681			
CC	0.541	0.702	0.656	0.662		
INFO	0.260	0.208	0.249	0.039	0.039	

V. Conclusions

The use of software quality metrics has become more common in the computer science community. Up to this point, however, their use has been mainly restricted to the measurement of production software or, in the case of design, limited to code metrics. This research is an attempt to show that (1) structure and hybrid metrics are extremely useful at design time, and (2) automatic generation of metrics for design specifications is not only possible with an analyzer similar to the one used for this research, but also desirable.

The results of this research clearly indicate several things:

- It is possible to generate meaningful complexity values at design time automatically.
- It is possible to predict the complexity values of the resultant source from the design measurements.
- Structure metrics are independent of level of refinement.
- Code metrics are NOT independent of level of refinement.

There is still much work to do with the idea of automatically analyzing designs. More data needs to be collected in order to verify and extend this work to incorporate more programs of radically different types, such as operating systems, compilers, etc. It would also be interesting to include several other metrics in the Software Metric Generator and compare the results with those of the metrics already generated. A current research effort under way at Virginia Tech is applying Software Quality Metrics in a graphical design environment.

New tools, such as the analyzer used in this research, must continue to be studied in an attempt to make the design process both more structured and more reliable. The incorporation of existing tools, such as software quality metrics, can be of great assistance in this undertaking and should be examined to determine their role in the software development process. Hopefully, this research is one of the first steps in the on-going search for better design methodologies.

VI. References

[BASV 75] Basili, V.R., Turner, A.J., "Iterative Enhancement: A Practical Technique For Software Development," IEEE Transactions on Software Engineering, Vol. SE-1, No. 4, Dec. 1975, pp. 390-396.

[BASV 83] Basili, V.R., Selby, R.W., Phillips, T.Y., "Metric Analysis and Data Validation Across Fortran Projects," IEEE Transactions on Software Engineering, Vol. SE-9, No. 6, Nov. 1983, pp. 652-663.

[BROF 86] Brooks, F.P., "No Silver Bullet: Essence and Accidents of Software Engineering," Information Processing '86, Elsevier Science Publishers B.V., 1986.

[CANJ 85] Canning, J.T., *The Application of Software Metrics to Large-Scale Systems*, Ph.D. Thesis, Computer Science Department, Virginia Polytechnic Institute, April 1985.

[CONS 86] Conte, S.D., Dunsmore, H.E., Shen, V.Y., *Software Engineering Metrics and Models*, The Benjamin/Cummings Publishing Company, Inc., 1986.

[ELSJ 78] Elshoff, J.L., "An Investigation Into The Effect of The Counting Method Used on Software Science Measurements," ACM SIGPLAN Notices, Vol. 13, No. 2, Feb. 1978, pp. 30-45.

[HALM 77] Halstead, M.H., *Elements of Software Science* , New York, Elsevier North-Holland, 1977.

[HENS79] Henry, S.M., *Information Flow Metrics for the Evaluation of Operating Systems' Structure*, Ph.D. Thesis, Computer Science Department, Iowa State University, Ames, Iowa, 1979.

[HENS 81a] Henry, S.M., Kafura, D., "Software Structure Metrics Based on Information Flow," IEEE Transactions on Software Engineering, Vol. SE-7, No. 5, Sept. 1981, pp. 510-518.

[HENS 81b] Henry, S.M., Kafura, D., Harris, K., "On the Relationships Among Three Software Metrics," Performance Evaluation Review, Vol. 10, No. 1, Spring, 1981, pp. 81-88.

[HENS 83] Henry, S.M., "A Project Oriented Course On Software Engineering," ACM SIGCSE Bulletin, Vol. 15, No. 1, Feb. 1983, pp. 57-61.

[HENS 84] Henry, S.M. and D. Kafura, "The Evaluation of Software Systems' Structure Using Quantitative Software Metrics," Software: Practice and Experience, Vol 14., No. 6, June 1984, pp.561-573.

[HENS 88] Henry, S.M., "A Technique for Hiding Proprietary Details While Providing Sufficient Information for Research," Journal of Systems and Software, 1988 (to appear).

[KAFD 82] Kafura, D., Henry, S.M., "Software Quality Metrics Based on Interconnectivity," Journal of Systems and Software, Vol. 2, 1982, pp. 121-131.

[KAFD 84] Kafura, D., Canning, J., Reddy, G., "The Independence of Software Metrics Taken at Different Life-Cycle Stages," Proceedings: Ninth Annual Software Engineering Workshop, NASA Goddard Space Flight Center, Nov. 1984.

[KAFD 85] Kafura, D., Canning, J., "A Validation of Software Metrics Using Many Metrics and Two Resources," Prodeedings: International Conference on Software Engineering, London, England, Aug. 1985.

[LIH 87] Li, H.F., Cheung, W.K., "An Empirical Study of Software Metrics," IEEE Transactions on Software Engineering, Vol. SE-13, No. 6, June 1987, pp. 697-708.

[MCCC 78] McClure, C., "A Model for Program Complexity Analysis," Proceedings: 3rd International Conference on Software Engineering, May 1978, pp. 149-157.

[MCCT 76] McCabe, T.J., "A Complexity Measure," IEEE Transactions on Software Engineering, Vol. SE-2, No. 4, Dec. 1976, pp. 308-320.

[REDG 84] Reddy, G., *Analysis of a DataBase Management System Using Software Metrics*, M.S. Thesis, Computer Science Department, Virginia Polytechnic Institute, June 1984.

[SELC 87] Selig, C.L., *ADLIF — A Structured Design Language for Metric Analysis*, M.S. Thesis, Computer Science Department, Virginia Polytechnic Institute, Sept. 1987.

[SHES 81] Sheppard, S.B., Kruesi, E., Curtis, B., "The Effects of Symbology and Spatial Arrangement On The Comprehension of Software Specifications," IEEE, 1981, pp. 207-214.

[TROD 81] Troy, D.A., Zweben, S.H., "Measuring the Quality of Structured Design," ACM SIGSOFT sponsored Software Engineering Symposiom, June 1981.

[WALR 78] Walpole, R.E., Myers, R.H., *Probability and Statistics for Engineers and Scientists, 2 Edition*, McMillan : New York, 1978.

[WOOS 80] Woodfield, S.N., *Enhanced Effort Estimation by Extending Basic Programming Models to Include Modularity Factors*, Ph.D. Thesis, Computer Science Department, Purdue University, Dec. 1980.