Design Metrics Which Predict Source Code Quality

Sallie Henry
Calvin Selig

TR 87-32

Abstract

Since the inception of software engineering, the major goal has been to control the development and maintenance of reliable software. To this end, many different design methodologies have been presented as a means to improve software quality through semantic clarity and syntactic accuracy during the specification and design phases of the software life cycle. On the other end of the life cycle, software quality metrics have been proposed to supply quantitative measures of the resultant software. This study is an attempt to unify the concepts of design methodologies and software quality metrics by providing a means to determine the quality of a design before its implementation. By knowing (quantitatively) the quality of the design, a considerable amount of time and money can be saved by realizing design problems and being able to correct these problems at design time. All of this can be accomplished before any effort has been expended on the implementation of the software. This paper provides a means of allowing a software designer to predict the quality of the source code at design time. Actual equations for predicting source code quality from design metric values are given.

Index Terms

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 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 quality metrics seems to indicate that such a comparison is not only possible, but also valid.

A typical software life cycle consists of requirements definitions, program design, implementation, testing, and finally maintenance [15]. 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.

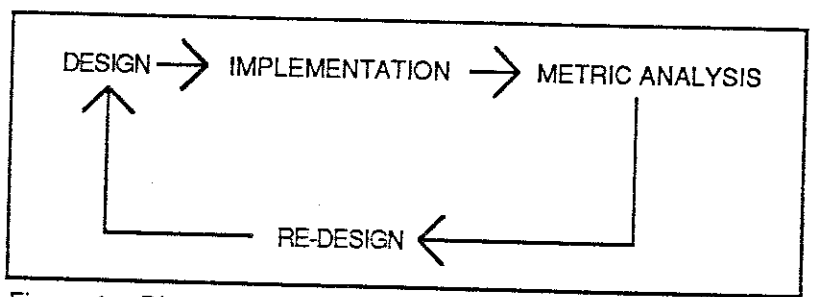

Figure 1. Diagram of Currently Used Software Life Cycle With Metrics

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 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 [17].

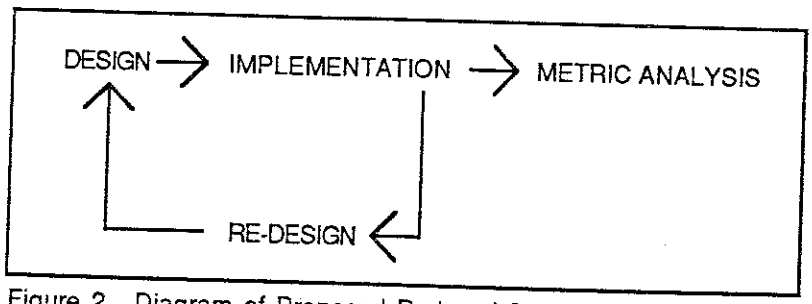

Figure 2. Diagram of Proposed Reduced Software Life Cycle With Metrics

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 quality 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 either the design or the source code as input and produces a number of complexity metric values as output. A diagram of the metric generation process is shown in Figure 3. For a more detailed explanation of the metric generator, see [10].

Many software quality metrics have been developed in recent years [5] [6] [13] [14] [19] [21] to name just a few. 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 three

general categories: code metrics, structure metrics, and hybrid 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 logic or semantics of the source program.

Finally, hybrid metrics consist of one or more code metrics combined with one or more structure metrics. 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 [8].

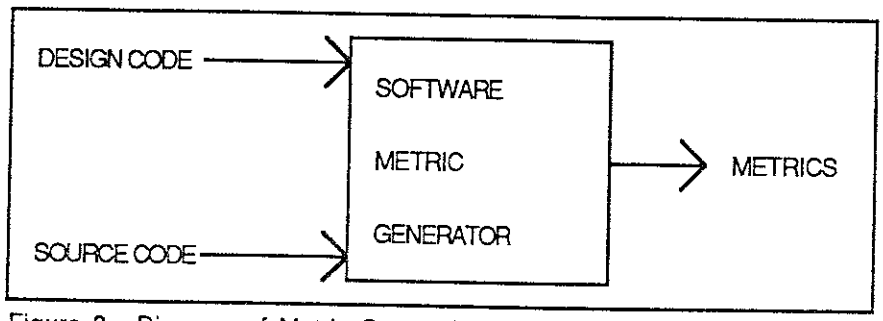

Figure 3. Diagram of Metric Generation Process

The next section briefly describes the software quality metrics used in this research. Section III presents the data used in this experiment. Analysis of the data as well as the equations generated to predict source code quality are given in section IV. Finally, a summary of our conclusions are given in section V.

II. Description of Metrics

This section contains brief descriptions of the metrics used in this research. Readers wishing greater detail should see the references.

A. 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 is widely used and has been extensively validated [2] [4] [16] [17].

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). Unfortunately, there is no consensus on exactly what constitutes a line of code. Most researchers agree that a blank line should not be counted but cannot agree on comments, declarations, null statements such as the Pascal "begin," etc. Another problem arises in free format languages which allow multiple statements on one textual line or one executable statement spread over more than one line of text.

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 [5]) 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_1 = the total number of operands.

N_2 = the total number of operators.

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, i.e., $N = N_1 + N_2$.

V represents the number of bits required to store the program in memory. Given n as the number of unique operators plus the number of unique operands, i.e., $n = n_1 + n_2$, then $log_2 (n)$ is the number of bits needed to encode every token in the program. Therefore, the number of bits necessary to store the entire program is:

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

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 difficulty is estimated as:

$$D = \frac{n_1}{2} \times \frac{N_2}{n_2}$$

Given V and D, the effort is calculated as:

$$E = V \times D$$

The unit of measurement of E is elementary mental discriminations which represents the difficulty of making the mental comparisons required to implement the algorithm.

McCabe's Cyclomatic Complexity

McCabe's metric [14] 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:

$V(G) = E - N + 2$ 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*.

B. 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's Information Flow Metric

Henry and Kafura [6] [7] 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.

To calculate the fan-in and fan-out for a procedure, a set of relations is generated that reflects the flow of information through input parameters, global data structures and output parameters. From these relations, a flow structure is built that shows all possible program paths through which updates to each global data structure may propagate [11].

The complexity for a procedure is defined as:

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

In addition to procedural complexity, the metric may be utilized for both a module and a level of the hierarchy of the system. Module complexity is defined as the sum of the complexities of the procedures in the module, and the level complexity is the sum of the complexities of the modules within the level.

C. Hybrid Metric

Since, as stated above, code and structure metrics appear to be measuring different aspects of program complexity, it seems reasonable that a metric be comprised of both types of metrics in order to capture the complexity of the procedure as much as possible. This is what is termed a hybrid metric. More succinctly, a hybrid metric is composed of one or more code metrics and one or more structure metrics. This study examines the hybrid form of Henry and Kafura's Information Flow metric.

Henry and Kafura's Information Flow Metric

The hybrid form of Henry and Kafura's Information Flow metric which was used in an actual study on the UNIX operating system is described in [7]. The formula is:

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

C_{ip} is the internal complexity of procedure p.

The metric used for the internal complexity Cip may be any code metric.

III. Description of Data

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 those generally obtained under these conditions.

Over the past several years, PDL designs and the resultant Pascal source code have 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 [9].

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 an ADA-like 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 is impossible to monitor all of the designs of all of the students; in fact, the students alone decide 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. Analysis of Data

The first step in the statistical analysis of the data is the elimination of outliers. In order to effectively eliminate invalid data while maintaining the overall integrity of the data, there are two major considerations. First is the effect of a change in the design that is not reflected in the given specifications but does appear in the resultant source code, and second is when specifications are not developed at a detailed enough level to make measurements meaningful. Note that the intent here is not to imply that a good design requires code-like specifications, but that a minimal amount of detail is necessary in order to have useful measurements. The first is evident when the complexity of the specifications for a procedure is larger than that of the source. It seems obvious that the complexity of the design should always be less than or, at best, equal to that of the source. When this is not the case, a design modification must have been made that does not appear in the source code. So as to not eliminate cases where the specifications are only slightly larger, only cases where the Information Flow complexity of the design is more than twice that of the source are eliminated. The second case appears when the source complexity is substantially larger than that of the specifications. For purposes of this study, "substantially" is defined to be source Information Flow complexity more than .one hundred times greater than the values for the design. After elimination of outliers using the above criteria, there are 981 modules left to analyze from 27 projects. Figure 4 shows the correlations of PDL to Pascal for all of the modules and metrics after the outlier elimination.

A. Analysis by Project

It is informative to note that in general, the Information Flow Metric in both its structure and hybrid form performs better than the code metrics. In fact, nine of the projects (2, 7, 11, 18, 19, 23, 24, 26, 27) have substantially lower correlations for the code metrics. A reason for this phenomenon is that the specifications are not well refined, but include those elements

necessary to the Information Flow Metric including procedure declarations, procedure calls, and updates to global variables. Project 23 is a good example of this general category of projects. The specifications for this project consist almost exclusively of those elements listed above. This type of design gives no clues as to the final form of the module. Therefore, it is impossible to predict the procedure complexity with the code metrics. There simply is not enough information in the design.

Project	Correlations								
NUM	LOC	N	V	E	CC	INFO	INFO-L	INFO-E	INFO-CC
1	0.781	0.828	0.891	0.753	0.670	0.940	0.992	0.981	0.829
2	0.325	0.335	0.412	0.117	0.524	1.000	1.000	0.999	0.999
3	0.364	0.387	0.357	0.019	0.390	0.523	0.254	-0.005	0.508
4	0.931	0.868	0.922	0.777	0.117	0.999	1.000	1.000	0.951
5	0.913	0.794	0.829	0.829	0.707	0.992	1.000	0.992	0.977
6	0.876	0.889	0.926	0.901	0.057	0.982	1.000	1.000	0.995
7	0.151	0.092	0.075	0.033	0.008	0.507	0.726	0.696	0.764
8	-0.059	-0.014	-0.015	-0.098	0.220	0.238	0.135	-0.035	-0.008
9	0.700	0.668	0.650	0.717	0.339	0.785	0.801	0.700	0.938
10	0.493	0.456	0.489	0.765	0.149	0.436	0.143	0.138	0.004
11	0.313	0.171	0.192	-0.040	0.631	0.954	0.827	0.106	0.959
12	0.593	0.511	0.557	0.007	0.421	1.000	1.000	1.000	1.000
13	-0.797	-0.376	-0.562	-0.998	-0.500	0.120	-0.183	0.646	-0.196
14	-0.075	-0.019	0.034	-0.056	0.053	0.306	-0.043	-0.088	-0.006
15	0.275	0.126	0.151	0.014	0.336	0.255	0.236	0.095	0.170
16	0.683	0.813	0.824	0.575	0.554	0.912	0.927	0.964	0.808
17	0.804	0.756	0.767	0.475	0.813	0.593	0.606	0.728	0.753
18	-0.135	-0.254	-0.162	-0.062	-0.120	0.639	0.718	0.848	0.952
19	0.324	0.329	0.301	0.091	0.218	0.766	0.326	0.696	0.562
20	0.799	0.681	0.722	0.635	NA	1.000	1.000	1.000	1.000
21	0.383	0.481	0.511	0.536	0.610	0.428	0.483	0.877	0.704
22	0.216	0.190	0.313	0.563	0.435	0.371	0.304	0.072	-0.017
23	0.621	0.548	0.615	0.202	0.573	0.997	0.998	0.984	0.907
24	0.348	0.595	0.764	0.548	0.574	0.837	0.789	0.687	0.855
25	0.800	0.897	0.918	0.946	0.857	0.717	0.741	0.841	0.828
26	0.177	0.340	0.341	0.370	0.392	0.704	0.839	0.892	0.907
27	-0.244	-0.281	-0.275	-0.172	-0.149	0.854	0.456	0.357	0.539

Figure 4. Correlations for all Modules Using all Nine Metrics

Seven of the projects (3, 8, 10, 14, 15, 21, 22) have poor correlation values for all of the metrics. This is probably due to poor design. Examination of the design code shows that the

designers consistently did not include any detail in the procedures. Not enough detail is provided to predict the code metric values, and the elements necessary for Information Flow are not available. In fact, within the above named projects, most of the PDL procedures closely resemble the one shown in Figure 5. Unfortunately, this is syntactically correct PDL code even though the semantic value is practically nonexistent. Once again note that a high level design is not necessarily a poor design, but it is much more difficult, if not impossilbe, to obtain useful measures.

```
PROCEDURE Example;
(This is an example of a poorly specified procedure.);
BEGIN Example;
  -- A comment explaining the function of the procedure.
END Example;
```

Figure 5. Example of a Poorly Specified Procedure

Two rows of Figure 4 that are of particular interest are rows 13 and 20. Row 13 contains values that appear to show an extremely high negative correlation. Obviously this is an undesirable result since it implies that the more complex the design code becomes, the simpler the source code is and vice-versa. Upon inspection, the reason for the unusual correlations becomes obvious. After outlier elimination, there are only three routines left to analyze. Project 13 is one of those that is not specified very well and really belongs to the group of procedures with all poor values. The small amount of useable data simply distorted the results. Row 20 is of interest because one of its entries in the Figure is "NA." This is due to the fact that after outlier elimination, all of the McCabe values in the design that are left have a complexity of 1. It is not possible to do correlations on a constant vector, and therefore that correlation is "Not Available."

The simple use of correlations on a project-by-project basis is not very informative as far as the ability to predict source code complexity is concerned. Many of the design teams typify a problem in software development. Given a choice between doing a job well and doing the same job with as little effort as possible on their part, they choose the latter. This problem reinforces the belief that the specification phase of a project must be closely monitored to ensure that a useful design is the end result. The ability to measure the designs should improve the performance of the designers.

B. Analysis by Level of Refinement

Since the results of the above correlations give so little useful information, it is informative to look at the data as a single entity as opposed to 27 different segments. It is hoped that by doing so, the projects that are not well specified will be offset by those that are. It is also 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 [17]. A highly refined module would contain code-like specifications, while a low refinement level indicates the predominance of natural language specifications. This should not be confused with the concept of high vs. low level **design** where a high level design indicates very little (if any) actual code. 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. Finally, several scatter plots are presented that allow a graphic representation of the data.

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 the refinement level increases, the length of the module also increases, but more importantly, the number of control structures will increase as well. Also, it appears necessary to consider *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 41 and 45. The normalizing function used in this study is:

$$Metric_{Norm} = \frac{(Metric_{Source} - Metric_{Design})}{(Metric_{Source} + Metric_{Design})}$$

Note that the normalizing function computes a value between zero and one as long as design complexity is less than or equal to the source complexity, and that the value approaches zero as the design metric value approaches the source metric value.

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 that determines the completeness of the design procedes as follows: First a normalized value is calculated for the McCabe Cyclomatic Complexity (CC) and Halstead's N metrics for each module. In the next step of the algorithm, the level of refinement is

determined using the normalized values. If the CC_{Norm} value is 0.0, then it can not be used to determine level, and the metric N is the sole determiner. If N_{Norm} is <= 0.333, then the level of refinement is high. If the value is between 0.333 and 0.667, then the refinement level is middle. Otherwise, the refinement level is low. If CC_{Norm} is not zero, the CC_{Norm} value is checked first, and if it appears fully refined (<= 0.333) the level is either high or middle, depending on N_{Norm}. If N_{Norm} <= 0.333, then the level of refinement is high and if it is > 0.333, the refinement level is middle. If the CC_{Norm} value is of middle refinement, the level is either middle or low, again depending on the value of N_{Norm} using 0.667 as the cutoff point. Finally, if CC_{Norm} indicates a low level of refinement, the module is determined to be low.

In order to determine the validity of the results of the algorithm, seven projects were examined and module 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 the same 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 modules. 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.

The analysis results in Figure 6 show that the above expectations are generally true. In the Figures, 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.

Figures 7, 9, and 11 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 Figures may be used to form an environmentally specific prediction equation, given that the level of refinement can be determined, with a 95% degree of confidence. Figures 8, 10, and 12 give the extra statistics needed to predict a confidence interval for a specific design complexity value and refinement level using the formula:

$$Slope \times X + Intercept \pm 2 X \sqrt{MSE \times \left(\frac{1}{n} + \frac{(X - X_{Mean})}{SS_X}\right)}$$ 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 the sum of squares over X and is given by:

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

[18].

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

Figure 6. Correlations by Level of Refinement

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:

$$1.126 \times 100 + 29.224 \pm 2 \times \sqrt{1.785E+5 \times \left(\frac{1}{276} + \frac{(100 - 11.388)}{1.490E+5}\right)}$$

$$141.824 \pm 55.100$$

In other 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.

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

Figure 7. Low Refinement Regression Line Equations and Statistics

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

Figure 8. Extra Statistics for Low Refinement Level

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

Figure 9. Middle Refinement Regression Line Equations and Statistics

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

Figure 10. Extra Statistics for Middle Refinement Level

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

Figure 11. High Refinement Regression Line Equations and Statistics

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
NFO	422	2.719E+17	1.650E+14	52433.400

Figure 12. Extra Statistics for High Refinement Level

In addition to the numerical statistics, it is informative to examine scatter plots as a means of graphically demonstrating the difference between code and structure metrics. To this end, Figures 13 and 14 show plots of the LOC metric and the Information Flow metric respectively, at a low level of refinement. The plots show the individual data points, regression line and the confidence interval. It must be noted that due to the large value of the Information Flow numbers, a log-log scale is used in the plots of that metrics' values. Figure 15 shows the LOC metric at a middle level of refinement indicating the improvement of the prediction. Figure 16 shows the Information Flow metric at a middle level of refinement demonstrating that the metric does equally well regardless of refinement.

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 based on the design complexity.

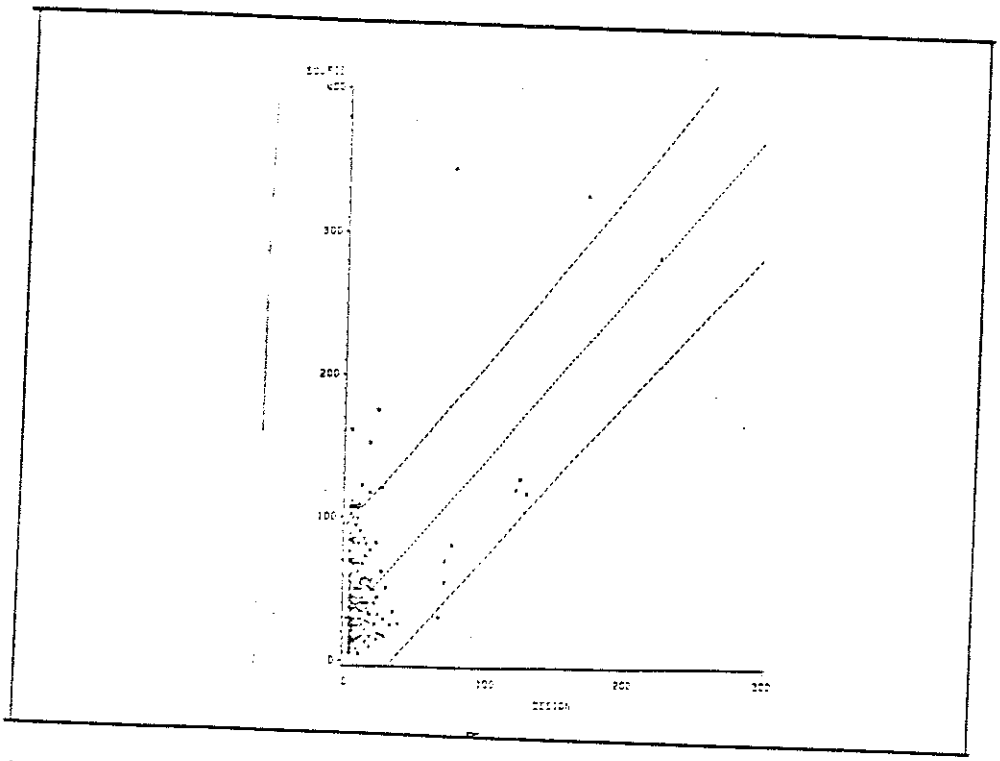

Figure 13. Low Refinement Regression and 95% Confidence lines for LOC

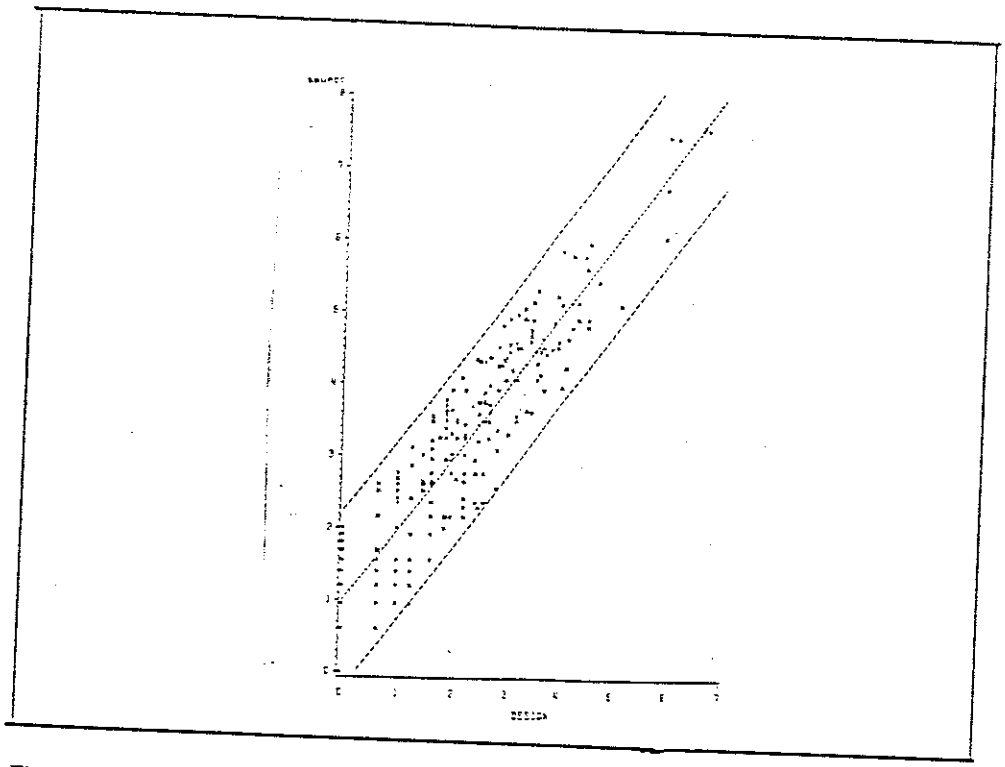

Figure 14. Low Refinement Regression and 95% Confidence lines for INFO

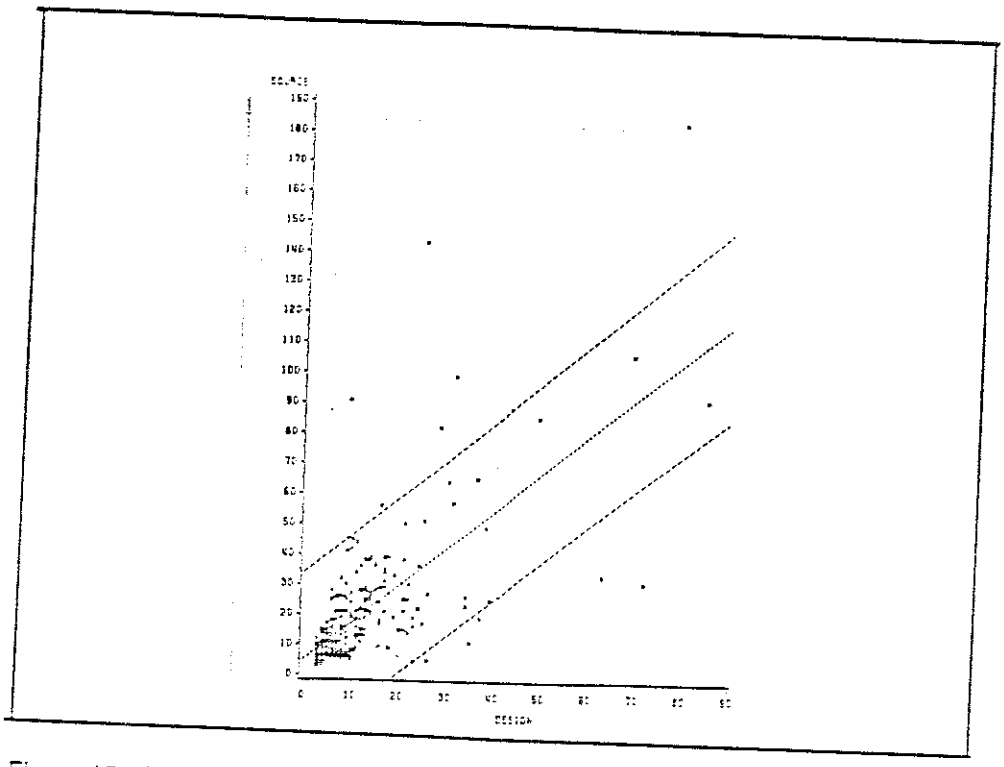

Figure 15. Mid Refinement Regression and 95% Confidence lines for LOC

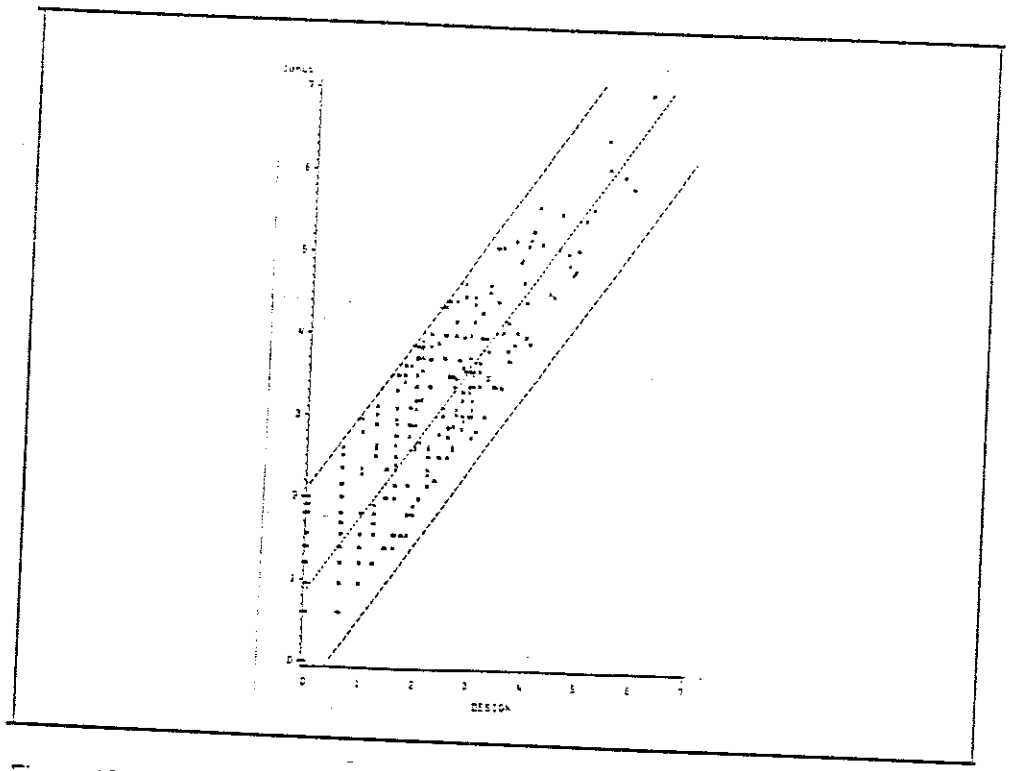

Figure 16. Mid Refinement Regression and 95% Confidence lines for INFO

C. Inter-metric Correlations

Figure 17 contains the inter-metric correlations for the Pascal code. This information serves to verify studies previously done by [1] [2] [8] [12] [16]. The figure 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 Figure 18. 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.

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	

Figure 17. Inter-metric Correlations for the Pascal 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	

Figure 18. Inter-metric Correlations for PDL code

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, (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 and (3) using the prediction equations presented in this paper and intimate knowledge of a design, a software designer can determine the level of refinement of a design and determine the complexity of the resultant source code.

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 [13] [20]. A current research effort under way at Virginia Polytechnic Institute 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

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

[2] 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.

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

[4] 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.

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

[6] 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.

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

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

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

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

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

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

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

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

[15] Overstreet, C.M., R.E. Nance, O. Balci, L.F. Barger, "Specification Languages: Understanding Their Role in Simulation Model Development," Virginia Polytechnic Institute Technical Report, SRC-87-001, December 1986.

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

[17] Selig, C.L., *ADLIF - A Structured Design Language for Metric Analysis*, M.S. Thesis, Computer Science Department, Virginia Polytechnic Institute, Blacksburg, Virginia, September 1987.

[18] Walpole, R.E., R.H. Myers, *Probabilityand Statistics for Engineers and Scientists, Second Edition*, McMillan : New York, 1978.

[19] Woodard, M.R., M.A. Hennell, D. Hedley, "A Measure of Control Flow Complexity in Program Text," IEEE Transactions on Software Engineering, Vol SE-5, No. 1, pp. 45, January 1979.

[20] Woodfield, S.N., *Enhanced Effort Estimation by Extending Basic Programming Models to Include Modularity Factors*, Ph.D. Thesis, Computer Science Department, Purdue University, West Lafayette, Indiana, December 1980.

[21] Yau, S.S., J.S. Collofello, "Some Stability Measures for Software Maintenance," IEEE Transactions on Software Engineering, Vol. SE-6, No. 6, pp. 545-552, November 1980.