A Design Tool Used to Quantitatively Evaluate Student Projects

Calvin Selig
Sallie Henry

TR 87-27

A Design Tool Used to Quantitatively Evaluate Student Projects

by

Calvin Selig
and
Sallie Henry

Computer Science Department
Virginia Tech
Blacksburg, VA 24061

Abstract

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, and statistics seem to indicate that this belief is true. Until recently, however, there existed no technique for quantitatively showing one program 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 is also valid.

The advent of Software Engineering has demanded that most universities offer a Software Engineering course which entails a "Real-World" group project. Students participating in the class design a system using a program design language (PDL). Oher students then write code from the design and finally the design team integrates the modules into a working system. For a complete description of the class see [HENS83] and [TOMJ87].

I. Introduction

In the area of software design, the basic problem is one of control. Without some form of control, the design process can degenerate into mass confusion consisting of disjoint ideas and goals. To help alleviate this problem, many varied methodologies have been formulated that allow, and even force the design process into a constructive and orderly state [BASV75] [DEMT82] [SHES81] [YOUE79]. In general, the design process consists of two separate processes. The first is general design, which encompasses such techniques as functional decomposition and top-down design, and the second is detailed design which includes flowcharts and pseudo code. A detailed design technique developed in the last few years involves the use of a program design language, often called simply a PDL. Design methodologies have ranged from flowcharts to functional decomposition to pseudo code, including PDLs.

Designers began to write code-like designs wherever possible to avoid the ambiguities of flowcharts. This technique became known as writing pseudo code. Designing in this style helped to reduce the communication gap between the programmers and the designers. Since there was no structured form for pseudo code, each individual used his own style and self-imposed syntax. It quickly became apparent that there must be some sort of a standardization in order to most efficiently use this technique.

By devising the concept of pseudo code with a specific syntax, the program design language came into existence. A program design language is different from a programming language since "while a programmer communicates with the computer, a designer communicates with people: managers, programmers, other designers, and himself" [SUTS81]. Due to this difference, the requirements for a PDL are somewhat different. A well designed PDL should have a rich typing environment as well as an adequate control structure set. It must, however, not be too restrictive. A designer using the language should have the ability to move outside the bounds of a stringent syntax and express himself in a natural language medium.

Other features desirable in a well constructed PDL should include: the ability to use the language at all levels of design from rough descriptions to detailed pseudo code, adaptability to the environment, familiarity to both designers and programmers, and susceptibility to the use of automated tools such as parsers, automatic document generators, and metric generators. A PDL with these features is indispensable as a design tool.

The concept of a program design language as a tool for specifications and requirements has become widely accepted, but it is not enough by itself. The need for design metrics is being acknowledged by some software organizations since PDLs do not provide any concrete, quantitative basis for making important design decisions. Metrical analysis of software is informative, but many times, "bad" code (according to the metrics) is caused by faulty decisions during design. It will be costly both in time and money to correct the design flaw. Ideally, the error should have been found much earlier in the program life-cycle. Design metrics appear to be the answer to some problems of this type.

Unfortunately, design metrics have not been extensively tested. Most papers, in fact, state that a specific metric could be used on their design methodology, but do not explain or validate the idea. In some cases where testing has been done, it is unclear if the metrics were automatically generated or hand generated. Finally, with the exception of the analyzer used in this study, all other proposed tools generate only code metrics [TROD81][SZUP81].

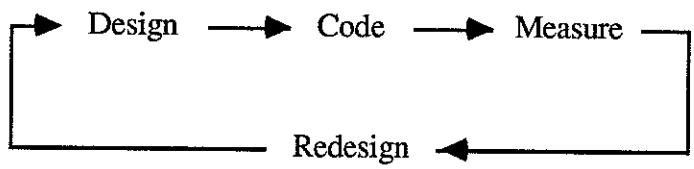

Figure 1. Standard Lifecycle

In general, the software life cycle starts at the requirements phase, then the design of the program, implementation, testing, and finally maintenance. The portion of the cycle [OVEC86] 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.

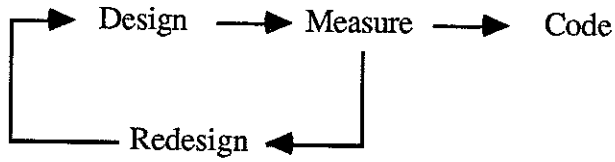

Figure 2. Reduced Lifecycle

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 generator, which for purposes of this study may be considered a "black box," is provided that takes as input either the design or the source code and produces, as output, a number of complexity metric values.

This paper presents a tool used by Software Engineering students to quantitatively evaluate their design. An overview of the PDL used in this study, ADLIF, is given in Section II. Section III presents the tool along with the Software Quality Metric generated by the tool. Our observations of the students' use of the tool and the students' designs are provided in Section IV and finally a summary of our conclusions is given in Section V.

II. The PDL, ADLIF

ADLIF is an acronym for Ada-like Design Language based on Information Flow. Its primary purpose is to create an intermediate step between the general design and coding phases in the life cycle of a system. It is NOT intended to be used as a programming language. Using ADLIF allows complexity measures to be taken before the code phase of a system. In this way, the quality of the software may be improved at an early stage, reducing time, man-hours, and cost invested in the system.

Since ADLIF is a design language and not a programming language, many actual statements in the system may be omitted or included in comment form instead of explicitly indicated. Therefore, in the most general sense, an ADLIF program may be mostly comments, specifically showing only variable declarations, subprograms with their parameters, subprogram calls with parameters, and updates of data structures. In some situations, it is more informative for a design to consist mostly of English sentences. At other times, in the most detailed use of ADLIF, pseudo instructions that are close to actual code are more desirable. ADLIF allows the designer to choose from either of these options, or anywhere in between. It seems reasonable that a more detailed (or code-like) design will result in more meaningful metric measurements. However, in most cases, the choice to design somewhere between English and code is most useful, since a more detailed design will not enhance understandability. This implies that an innately complex piece of software will require a more code-like design reflecting the fact that the software is not readily understandable to the programmer. ADLIF contains most of the basic programming statements and these may be used at the designer's discretion for specific algorithms or formulas.

This section uses portions of the reference manual for the language ADLIF to discuss the more interesting or complex portions of the language. Questions regarding specific syntax are refered to [SELC87].

Comments

ADLIF contains three different types of comments. The first is the general comment typical of most programming languages to allow user documentation. This is a string of characters preceded by two hyphens that continues until the end of the line, where an automatic termination of the comment occurs. This type of comment is used for general documentation.

```
-- THIS IS A COMMENT.
C :=      -- Comments are ignored when
A + B     -- occuring any place in a program.
```

The second type of comment is a required comment, and is enclosed in parentheses and followed by a semicolon. Comments are required following the program statement and after a subprogram declaration. The intent is to ensure that each subprogram, as well as the overall design, is documented.

The final type of comment in ADLIF is one enclosed in braces. This type of comment may be used in a statement as a replacement of a mathematical expression. For example: ({use quadratic formula} + 10), where the comment constitutes either some or all of the expression. It enables the designer to leave the details of expression generation up to the programmer by allowing "pseudo" expressions to be substituted for an actual expression, when use of the comment will simplify or clarify some aspect of the design.

Comments are descriptive and may be used anywhere in a program except before the program declaration and following the "END program-name" statement.

Built-in Functions and Procedures

Some built-in functions have been provided in ADLIF and may be assumed to exist. Since ADLIF is not a programming language, it is only necessary that the syntactic analyzer recognizes the existence of these built-in routines. These functions constitute the normal set of supplied functions in most programming languages. Their inclusion in ADLIF is simply to allow more design flexibility. They include: ROUND, TRUNC, INTTOREAL, EOLN, EOF, NEW, ORD, CHR, SUCC, and PRED.

Parameter passage

In a procedure, parameters may have one of three different modes. The modes available are those defined in Ada. They allow the designer to determine how a parameter may be used within a procedure: either allowing a value to be passed in only (pass by value), passed out only, or both. In a procedure declaration, the mode of a parameter occurs between the variable name and the variable type.

The three parameter modes available for procedures and their meanings are:

IN — The parameter acts as a local constant whose value is provided by the corresponding actual parameter. If a mode is not specified, IN is assumed.

OUT — The parameter acts as a local variable whose value is assigned to the corresponding actual parameter as a result of the execution of the subprogram.

INOUT — The parameter acts as a local variable and permits access and assignment to the corresponding actual parameter. A value must be provided by the corresponding actual parameter, and a value is assigned to the corresponding actual parameter as a result of the execution of the subprogram.

When using a function, a mode is not allowed: and the assumed mode is IN. This forces functions to return a single value using the function name. If the designer feels that it is required to use a mode other than the default of IN, the function must be transformed into a procedure. This follows the mathematical definition of a function.

Packages

A facility called "Packages" is supplied in ADLIF in order to have the capability of modularizing a program design. Using a package, subprograms may be designed and used that are completely separate from any specific ADLIF program, thus allowing the use of the information hiding concept. The overall form of a package is similar to that of an ADLIF program, and is given formally below:

```
PACKAGE package-name IS      --  This is the package
                             --  specification, and is
Visible-entities;            --  the only part of the
                             --  package visible to the
END package-name;            --  outside world.
PACKAGE BODY package-name IS
(comment);
[Declarative part;]          --  This is the body of the
                             --  package and contains
BEGIN package-name           --  all code related to the package.
statement[s];
END package-name;
```

In the specification above, package-name is any valid unique ADLIF name. Visible-entities are similar to a normal Declarative part, with the exception that only those constants, types, variables, and subprogram declarations that are to be visible to the calling entity should be specified. Anything that is to be hidden to the calling entity should be omitted from this section. Note that procedures and functions not shown in the visible-entities section may only be used internally in the package. The declarative part is identical to that in an ADLIF program or subprogram, as are statement[s]. Note that subprogram declarations shown in the visible-entities section must be complete and identical to the declarations repeated in the declarative section in the body of the package.

In order to make the package specification visible to a program or another package, the reserved word USE is needed and is placed as the first statement in the declarative section, with the following syntax:

```
USE package-name [,package-name];
```

Each declaration in the visible-entities section of the package must then be duplicated in the declarative section of the program using the package, with each entity followed by a package reference which indicates where the entity may be found. See below for the specific syntax of a package reference.

Visible-entity; PACKAGE package-name;

Each of the constants, types, variables, and subprograms from the package may be used or called in the normal manner.

Due to the similarity to Ada and Pascal, ADLIF is both simple to learn and relatively easy to use. The use of programming constructs allows the designer the flexibility to decide how much of the design needs to be in pseudo code and how much may be in English sentences. Both the ease of use and the flexibility, along with the ability to do complexity measures early in the program life cycle, should make ADLIF a valuable design tool.

III. A Description of the Tool

The tool used in the Software Engineering class is called the Software Quality Metric Analyzer. Input to the analyzer is Pascal, "C", THLL (a language used by the Navy), or ADLIF code. ADLIF is the only design language currently available. Software Quality Metrics corresponding to the design of source code are output. Figure 3 gives a simplistic view of the analyzer. The analyzer is based on LEX (a lexical analyzer) and YACC (Yet Another Compiler - Compiler) which are tools available with a UNIX environment. For a complete description of the analyzer see [HENS88].

The remainder of this section describes the metrics, lines of code, McCabe's Cyclomatic Complexity and Halstead's Software Science measures, and Henry and Kafura's Information Flow.

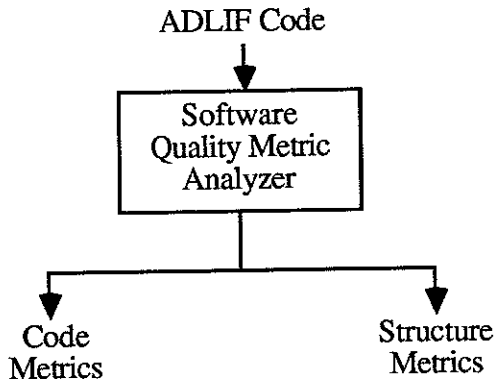

Figure 3. A Simple View of the
Software Quality Analyzer

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 are widely used and have been extensively validated [CANJ85] [ELSJ78] [REDG84][HENS81b].

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: 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 program headers, declarations, executable and non-executable statements.

Halstead's Software Science

A natural weighting scheme used by Halstead in his family of metrics (commonly called Software Science [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_1 = the total number of operands.

N_2 = the total number of operators.

Three of the software science metrics will be discussed, specifically N, V, and E.

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., $N = N_1 + N_2$. An operand is defined as a symbol used to represent data and an operator is any keyword or symbol used to express an action [CONS86].

The second Halstead metric considered is volume (V). 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 [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 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.

Structure Metrics

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 between 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 inter-dependencies involved contribute to the overall complexity of the program units, and ultimately 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 [HENS79] [HENS81a] 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.

The complexity for a procedure is defined as:

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

where

C_p	=	the complexity of procedure p
fan-in	=	the number of fan-ins to procedure p
fan-out	=	the number of fan-outs from procedure p.

The term *(fan-in x fan-out)* is squared to represent the idea that the complexity due to the inter-relationship between components is non-linear.

In addition to procedural complexity, the metric may be utilized for both a module, which consists of those procedures which either update or retrieve information from a global data structure, 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.

Procedural complexity is used to find those procedures with heavy data traffic and those not adequately refined. Module complexity reveals overloaded data structures as well as improper modularization. The level complexity may be used to detect missing levels of abstraction or to compare alternate system designs.

IV. Observations

Over the past several years, ADLIF designs and the resultant Pascal source code have been collected from undergraduate, senior-level Software Engineering courses at both Virginia Polytechnic Institute and the University of Wisconsin--LaCrosse.

The completed projects that have been furnished by the classes are varied. They range from two thousand to eight thousand lines of code. Due to the fact that the experience is the important goal of the assignment and not final program function, many groups have developed games such as: chess, Othello, backgammon, and Monopoly. Not all of the teams, however, chose this route. Other types of projects have been developed, including a hierarchy chart generator, a Pascal code formater, a calendar system, and a bulletin board.

In an attempt to ensure that the completed designs and source are usable as data, students are given minimal design requirements. The requirements are to include in their specifications:

1. Procedure and function calls.
2. Updates to global data structures.
3. Parameters necessary to each procedure.

Some tools have been developed to aid students in the specification portion of the project. An ADLIF hierarchy chart generator is available which allows the design teams and programmers to get an overall feel for the structure of the program and to more easily determine problems inherent with the calling hierarchy. There is also an ADLIF syntactic analyzer is used, which is useful in eliminating common errors such as typing errors, as well as determining missing components such as parameters, variable declarations, etc. The syntax analyzer also ensures that the projects, which are used as data, are in an analyzeable form. Finally, the software metric analyzer so that, by students to generate design metrics which can be examined to indicate possible trouble spots in the design such as inadequate refinement, poorly designed data structures, and missing levels of abstraction.

Unfortunately, due to time constraints in the class, the students are told only that relatively large metric values indicate a procedure with high complexity. Subsequent lectures in the class

describe the metrics in greater detail.

To date, only subjective comments from the student designers indicate the accuracy of the metrics. Students with detailed designs have concluded that the metrics actually indicate the "most difficult routines" in their systems. We stress the word "detailed" since many design teams choose to use more English-like specifications. Some of these specifications include only comments and a BEGIN-END statement. Obviously, most of the metrics can provide little insight into such an inadequately refined module.

Our observations concluded that the Structure Metric (Information Flow) was able to indicate complexity even in those routines with inadequate refinement. Since the Information Flow metric requires only calls, parameter passage and global variable use, its credibility can be shown at all levels of refinement. The code metrics, as the name implies, requires more refined routines before the metric values can be valid [SELC87].

V. Conclusions

ADLIF has been used by students for several years. Although the students protest over learning "yet another new language," they like the idea of measuring their designs. The Software Quality Metric Analyzer has only been used for one course, however, the students found that the tool successfully predicted which modules would be difficult.

In subsequent software engineering courses, it is our hope that the students use the metrics to redesign their system prior to coding and to provide them with direction in testing. As most educators have experienced, students seldom test their software sufficiently. With the additional insight of the metric information, the students may focus their testing strategies on the more complex procedures.

References

[BASV75] 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.

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

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

[DEMT82] DeMarco, T., *Controlling Software Projects*, Elsevier North-Holland Publishing Company, 1982.

[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, Feb. 1978, pp. 30-45.

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

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

[HENS81a] 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.

[HENS81b] 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.

[HENS83] Henry, S.M., "A Project Oriented Course On Software Engineering," ACM SIGCSE Bulletin, Vol. 15, No. 1, Feb. 1983, pp. 57-61.

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

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

[OVEC86] Overstreet, C.M., Nance, R.E., Balci, O., Barger, L.F., "Specification Languages: Understanding Their Role in Simulation Model Development," Virginia Polytechnic Institute Technical Report, SRC-87-001, Dec. 1986.

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

[SELC87] Selig, C. *ADLIF – A Structured Design Language for Metric Analysis*, M.S. Thesis, Computer Science Department, Virginia Tech, September 1987.

[SHES81] Sheppard, S.B., Kruesi, E., Curtis, B., "The Effects of Symbology and Spatial Arrangement On The Comprehension of Software Specifications," IEEE Computer, 1981, pp. 207-214.

[SUTS81] Sutton, S.A., Basili, V.R., "The Flex Software Design System: Designers Need Languages, Too," IEEE Computer, Nov. 1981, pp. 95-102.

[SZUP81] Szulewski, P.A., Whitworth, M.H., Buchan, P., DeWolf, B., "The Measurement of Software Science Parameters in Software Design," Communications of the ACM, March 1981, pp. 89.

[TOMJ87] Tomako, J. "A Class Project in a Software Engineering Course," presented at the Software Engineering Institute (SEI) Workshop, October 1987.

[TROD81] Troy, D.A., Zweben, S.H., "Measuring the Quality of Structured Design," ACM SIGSOFT sponsored Software Engineering Symposiom, June 1981.

[YOUE79] Yourdan, E., Constantine, L., *Structured Design*, Prentice Hall, 1979.