Measurement of Ada Throughout the Software Development Life Cycle

Bryan L. Chappell, Sallie Henry and Kevin Mayo

TR 89-44

MEASUREMENT OF ADA THROUGHOUT THE SOFTWARE DEVELOPMENT LIFE CYCLE

Bryan L. Chappell

Software Technology, Inc.
5904 Richmond Highway, Suite 610
Alexandria, Virginia 22303

Sallie Henry
Kevin A. Mayo
Computer Science Department
Virginia Tech
Blacksburg, Virginia 24060

Summary

Quality enhancement has now become a major factor in software production. Software metrics have demonstrated their ability to predict source code complexity at design time and to predict maintainability of a software system from the source code. Obviously metrics can assist software developers in the enhancement of quality. Tools which automatically generate metrics for Ada are increasing in popularity. This paper describes an existing tool which produces software metrics for Ada that may be used throughout the software development life cycle. This tool, while calculating established metrics, also calculates a new structure metric that is designed to capture communication interface complexity. Measuring designs written using Ada as a PDL allows designers early feedback on possible problem areas in addition to giving direction on testing strategies.

This research was funded by Software Productivity Consortium (SPC), and the Center for Innovative Technology (CIT).

KEYWORDS: Software Design, Software Design Tools, Software Metrics, Software Development Life Cycle, PDL Measurement.

Introduction

"You can't control what you can't measure" [DEMAT82], [YOURE89]. Interest in controlling the software development process has always been a concern for management but more recently it has received attention from all levels of software development personnel. The software crisis has made software developers more aware of the need to measure (or even better, to predict) the quality of their software product. The DOD has also taken steps to assure the quality of their contractor's source code. Several tools have been developed to generate metrics for various languages. Software metric researchers have realized that more time,

effort, and money can be saved if the measurement is performed throughout the software development life cycle, especially during the design phase. With the growing popularity and use of the Ada programming language, the next extension must be to analyze Ada.

Several studies have been done that support the use of measurement throughout the software life cycle. Ramamoorthy et al. [RAMAC85] suggest the use of metrics throughout the software life cycle, emphasizing that different metrics need to be used during different phases of software development. Kafura and Canning [KAFUD85] suggest the use of structure metrics in the design and development phase. Metrics applied in the design of a software system can be used to determine the quality of a design, and can also be used to compare different designs for the same requirement specifications [HENRS89], [HENRS90b], [HENRS90c].

Using metrics during the testing of a software product helps to determine the reliability of the software. Lew, Dillon, and Forward [LEWK88] used software metrics to help improve software reliability. The software metrics were able to quantify the design and provide a guide for designing reliable software.

As previously noted, maintenance efforts cost the most of any part of the software development life cycle. Kafura and Reddy [KAFUD87] used seven metrics in a software maintenance study. Among other results, they found that the growth in system complexity that was determined by the metric measurements agreed with the general character of the maintenance tasks that were performed. Other research shows that metrics applied during the coding phase can predict the maintainability of the software product [WAKES88], [LEWIJ89].

It is obvious from the preceding paragraphs that software metrics can be a useful guide in the life of a software product. In order to facilitate the use of software metrics, an automatable metric analysis tool must be developed. Gilb, in [GILBT77], stresses the importance of an automated metric analysis tool, citing a TRW study where metrics-guided testing is half the cost of conventional testing.

This research produced a software metric tool for Ada that can be used throughout the entire software development life cycle. Some of the metrics produced are based on the structure of the code, which can be seen early in the software design. This aspect, coupled with the encouragement by the Ada community to use Ada as a PDL, provides for the gathering of metrics at the design stage with the metric analysis tool.

There is a benefit of having a design language that is a subset of a programming language while also having design language features. Many researchers have proposed the use of Ada as a program design language (PDL) [CHASA82], [GABBE83], [SAMMJ82]. Each of the military services under the Department of Defense is currently issuing requests for proposals (RFPs) which require the use of an Ada-based PDL [HOWEB84].

The Software Metric Analyzer that was developed as part of this research can be used to analyze Ada code written with any level of refinement. Therefore, the analyzer supports the use of Ada as a PDL. Design quality can be determined and different designs for the same requirement specifications can be compared. Henry and Selig [HENRS90b] have been able to predict the complexity of the resultant source code from measurement of designs written in an Ada-like PDL. Many metrics are calculated by the Software Metric Analyzer, including the code and structure metrics that are discussed in the next section. The structure metrics produce more relevant information than code metrics when analyzing design code. This is because most of the details of the design code concerns the system's hierarchy and calling structure. The Software Metric Analyzer should be of great benefit as a life cycle support tool.

Metrics for Ada

Once it is realized that software needs to be measured, it is important that the measuring process be automatable. It would be unreasonable to consider measuring software manually due to the size of most software systems. Also, automatable measuring is more accurate and consistent. Therefore, this research concentrated on only those metrics that are quantitative and automatable. The available quantitative metrics can be grouped into three rather broad categories. These categories are *code metrics*, *structure metrics* and *hybrid metrics*. Code metrics produce a count of some feature of the source code. Structure metrics attempt to measure the logic and interconnectivity of the source code. Hybrid metrics combine one or more code metrics with one or more structure metrics. Code metrics are easier to calculate than the structure and hybrid metrics, but since the structure and hybrid metrics measure different aspects of the original source code, they are very important and worth the effort of computation [HENRS81a], [HENRS90b].

Code Metrics

Code metrics produce a count of some feature of the source code. The code metrics that this study uses are lines of code [CONTS86], Halstead's Software Science [HALSM77], and McCabe's Cyclomatic Complexity [MCCAT76].

Of the many definitions of lines of code (LOC) that are discussed in the literature, the definition that is used in our research is from Conte et al.:

> A line of code is any line of program text that is not a comment or blank line, regardless of the number of statements or fragments of statements on the line. This specifically includes all lines containing program headers, declarations, and executable and non-executable statements.

Halstead's Software Science Indicators are well-known and used frequently. Halstead's metrics are based on token counts of operators and operands. Therefore, part of this research involved defining what the operators and operands are for Ada. Of the many metrics proposed by Halstead, this research calculates his program length (N), program volume (V) and effort (E).

McCabe's Cyclomatic Complexity (CC) is based on the logic structure of subprograms and has its basis in graph theory. His metric measures the number of basic paths through a subprogram. McCabe's metric is calculated by using the shortcut he presented, i.e., counting the number of simple conditions and adding one.

Structure Metrics

Structure metrics attempt to measure the logic and control flow of a program. The structure metrics that this study incorporates are Henry and Kafura's Information Flow metric [HENRS81b] and interface metrics [MAYOK89].

Henry and Kafura's Information Flow metric (INFO) is based on the information flow connections between a subprogram and its environment. These connections can be in the form

of subprogram invocations and accesses to "global" data structures, i.e., those available through a common package specification.

Interface metrics attempt to more fully measure the interfaces among subprograms by recognizing several facts about software: variables of different types have a different complexity inherent in their types (e.g., an integer variable is less complex than a record variable), operations vary in complexity (e.g., addition is less complex than division), and conditionals vary in complexity (e.g., a FOR loop is different in complexity than a WHILE loop). Also, how variables are referenced (either they are read from or written to, or both) contributes to complexity. All of these facts combine to yield a better measure of the complexity of subprogram invocations.

Interface metrics were defined with the Ada programming language in mind. With the possibilities available through the use of packages and generic units, program complexity needs to be viewed in a new way [GANNJ86], [SHATS88].

Hybrid Metrics

Hybrid metrics consist of one or more code metrics combined with one or more structure metrics. The hybrid metrics that this research uses are the hybrid form of Henry and Kafura's Information Flow metric [HENRS81b] and Woodfield's Review Complexity (RC) [WOODS80].

Henry and Kafura's Information Flow metric in its hybrid form is a multiplication of any code metric and the structure form of the information flow metric. Woodfield's metric is associated with programmer time. He states that certain subprograms must be reviewed by a programmer several times due to their interconnections within the program. Woodfield defines three types of connections between subprograms: subprogram invocation, modification of global data, and assumptions made in one subprogram that are used in another subprogram. Woodfield's Review Complexity uses Halstead's effort (E) as the code metric, but suggests that any code metric could be used.

The Software Metric Analyzer

The software metric research group at Virginia Polytechnic Institute has developed a Software Metric Analyzer. A diagram of the Software Metric Analyzer is in Figure 1 (at the end of this paper).

The Software Metric Analyzer can be run using Ada code written at any level of refinement. Therefore, the analyzer supports the use of Ada as a PDL. This enables designers to analyze their design before it is completely implemented. Hopefully, design errors can be caught during the design and will not be allowed to propagate through to the implementation. There is a need for life cycle support in an Ada environment. The Software Metric Analyzer should be of great benefit as a life cycle support tool.

Phase 1

The main thrust of this research effort was in developing the Ada Translator for use in calculating the metrics discussed in the preceding section. The Ada Translator takes as input Ada source code or Ada PDL, as well as information on the pre-defined language environment (e.g., the package STANDARD). The Ada Translator is written in Lex, YACC, and C

[LESKM75], [JOHNS75]. The outputs from the translator are code metrics and relation language code. Phase 1 is the only phase to see the source code, thus, phase 1 generates the code metrics which rely on source code token counts.

The relation language that is produced by phase 1 is our own, in-house object language [HENRS88]. The Ada source code is translated into the equivalent relation language source which contains all the information that is necessary to generate structure and hybrid metrics.

Also, it is not necessary for the software producer to worry about loosing any proprietary information embedded within the Ada source code. The Ada Translator could be run at the producer's location and the code metrics file and the relation language code file could be processed afterwards at Virginia Polytechnic Institute. The translation from Ada to relation language, while producing sufficient information to generate structure and hybrid metrics, removes enough detail that there is no worry of loosing any proprietary information [HENRS88].

The Ada Translator keeps a project library, much like an Ada compiler does. This information is maintained solely by the translator for use in processing later compilation units. This helps minimize problems such as name resolution and symbol table processing across packages.

Phase 2

From the relation language code, interface metrics are calculated in their entirety. Also, relations are computed. Relations are grouped by subprogram and are used in phase 3. A set of relations is computed for each variable within a subprogram. Relations identify flow of control and flow of information throughout the system.

Phase 3

The third phase of the Software Metric Analyzer is where all the information produced thus far is accumulated and assembled into a format accessible to the user. The code metrics from phase 1 and the interface metrics from phase 2 are included in this final phase. With the relations generated in phase 2, the Metric Generator calculates structure metrics. The code metrics, together with these newly calculated structure metrics, allow the hybrid metrics to be calculated. Therefore, all metric values are available to the user from this third and final phase of the Software Metric Analyzer.

The analyzer has two different methods of grouping units of analysis. Modules can be defined in order to group a package together with all of its nested subprograms and packages. In this way, complexity analysis can be performed on larger sections of code. The other grouping mechanism of the analyzer is more flexible and allows any combination of units of analysis to be grouped together, regardless of their lexical nesting level.

The Data

Software Productivity Solutions (SPS) is a small company (less than 40 employees) interested in producing quality software products. Located in Melbourne, Florida, most of their contracts are from the Department of Defense. The data used in the analysis of the Software Metric Generator is an Ada-based Design Language (ADL) Processor produced by SPS. The system was donated to the Virginia Tech Metrics Research Group for use in the

validation of the Software Metric Analyzer. The ADL programming system provides for syntax checking and limited semantic checking of ADL source files. Also, it generates a variety of data dictionary and cross reference reports and has an on-line help facility. The system consists of four subsystems: support, database, report, and analyzer. In total, this system has 83,799 textual lines of code and 5,355 subprograms and packages. The support section contains 6,702 lines of code. The database section contains 43,019 lines of code. The report section contains 3,874 lines of code. The analyzer section contains 30,204 lines of code.

Ada Metrics Results

The Software Metric Analyzer processed the ADL Processor's source code and the resultant code, structure, and hybrid metrics were analyzed. All correlations in this paper are Pearson correlations.

After the data was gathered, the metrics had to be validated. Usually this is done by using error history analysis or development time information but none of this was available. Therefore, the validation was performed by a subjective hand inspection of some of the subprograms that were flagged as potential problem areas.

Those subprograms perceived as potential problem areas are the outliers. A subprogram is considered to be an outlier if *one* or more of its metric values falls in the top 1% of that metric's values. Since the data is clustered towards small complexity values, only the largest 1% of the data was removed. The range of metric values would definitely vary from system to system, therefore the definition of an outlier would need to vary as well.

The subprograms that were inspected were those that have high complexity values for all of the metric measurements. Table 1 contains information concerning the overlap of the outlier data among the metrics. See Table 2 for some summary statistics of the data and Table 3 for the intermetric correlations of the data with the outliers removed.

It can be seen from the information in Table 1 that 76% of the outliers are in the top 1% of *only one* metric's value range. Most of these subprograms probably should not be considered problem areas since only one metric considers them to have a high complexity. This is a good example of showing the need to use more than one metric in analyzing a programming system.

On the other end of the spectrum, 1.7% of the outliers are considered outliers based on *all* of their metric values. Surely these subprograms are the ones that should be examined carefully, tested rigorously or perhaps be redesigned.

The intermetric correlations of the data in Table 3 are consistent with previous results for languages other than Ada[HENRS81a], [LEWIJ89], [HENRS90b]. The code metrics correlate highly, which seems to agree with previous metric studies. McCabe's Cyclomatic Complexity metric does not correlate to the other code metrics as well as all of the other code metrics correlate to one another. Part of the reason for this could be the fact that package definitions are not required to have a body of code with them.

Table 1. Overlap of Outliers

Number of Metrics In Overlap	Percentage of the Outlier Data
1	76.0
2	5.5
3	2.5
4	2.9
5	5.6
6	5.8
7	1.7

Table 2. Summary Statistics of the Data

Metric	Mean	STD Dev	Minimum	Maximum
LOC	16.3	15	1	150
N	83.0	83	4	645
V	429.2	518	8	4055
E	11152.5	19049	12	184177
CC	2.4	2	1	18
RC	11506.6	19568	12	182989
INFO	12510577.0	67495181	1	900486748

Table 3. Intermetric Correlations of the Data

Metric	LOC	N	V	E	CC	RC
LOC						
N	0.947					
V	0.942	0.997				
E	0.916	0.957	0.965			
CC	0.668	0.726	0.717	0.727		
RC	0.904	0.940	0.946	0.968	0.700	
INFO	0.065	0.090	0.098	0.072	-0.007	0.106

Woodfield's Review Complexity correlates very high (0.968) to the code metric used (Halstead's *E*). This is because most of the units of analysis have a Woodfield *fan-in* value that is less than or equal to 3. This yields a Review Complexity value equal to Halstead's *E*. Packages could be part of the reason for Woodfield's *fan-in* not exceeding 3 that often. A package with only a package specification will never have a Woodfield *fan-in* that exceeds 3

because packages can not be called, nor can they modify a data structure that is declared external to them.

It can be seen in Table 3 that Henry and Kafura's Information Flow metric does not correlate at all to the other metrics. This is because the Information Flow metric measures the flow of information between units of analysis and does not measure size. The Information Flow metric is probably a better indicator of complexity than the size metrics, because although size contributes to complexity, module interactions make programs difficult to write, test and understand. These interactions are also the source of many errors [HENRS81a].

It should also be noted that since the Information Flow metric does not depend on size but rather on the flow of information, it can produce meaningful results from analyzing a design written in Ada or an Ada-like PDL. This is because designs written in a PDL are broken down by procedures and contain calls to other procedures. With just this calling structure, the Information Flow metric can compute a meaningful complexity measure [HENRS90b].

Interface Complexity Results

The findings from the interface complexity study are promising. There were 20,975 observations of intra-package communication. These complexities were reduced to form eight interface complexity measures associated with each procedure. These measures are:

Table 4: Interface Complexity Measures

AccN	The number of accesses from the current procedure to other procedures.
Acc	The measure representing the environment complexity at the time of the calls.
P1	The measure capturing the effect of parameter variable types on communication.
P2	The measure showing the effect of modification on the parameters.
FAccN	The number of accesses into the current procedure from external sources.
FAcc	The measure representing the environment within external procedures during calls to the current procedure.
FP1	The measure showing the effect of parameter variable types on communication within external calling procedures.
FP2	The measure showing the effect of modification to the parameters that are being sent into the current procedure from within external procedures.

These measures correlate with the established metrics in varying degrees. However, it is the function of these variables that interests this study. The following four functions were examined:

Table 5: Functions of Interface Complexity Measures

F1	(Acc * P1) + (FAcc * FP1)
F2	(Acc * P1) * (FAcc * FP1)
F3	(Acc * P2) + (FAcc * FP2)
F4	(Acc * P2) * (FAcc * FP2)

Of these functions, F2 and F4 correlate very high with the Henry-Kafura Information Flow. Therefore, why reinvent the wheel? However, function F1 showed moderate correlations to both code and hybrid metrics. This is interesting due to the fact that F1 is measuring factors that are similar to both code and hybrid metrics.

Table 6: Correlations among Functions and Established Metrics

Function	LOC	E	INFO
F1	0.28539	0.47745	0.50984
F2	0.03311	0.02996	0.91971
F3	0.34048	0.57726	0.06788
F4	0.03270	0.02950	0.91970

The lack of an error history compromised this study, and to compensate for this deficiency a systematic subjective examination of the code was undertaken. This involved rank ordering the eight interface complexity measures, and pulling the subprograms associated with these highest valued measures.

The metrics correctly measured their intended purposes. Furthermore, this inspection showed that these measures could be used to detect areas of the code (or design) with potential problems. These metrics also pointed out Ada-specific characteristics, such as a package of definitions. This package, while having no code, is full of declarations for variables. Therefore, any reference to one of these variables is a communication to the package. These metrics proved very useful in the detection of questionable programming styles, including data hiding.

Used at the design stage, these metrics could predict areas of abuse, and areas in need of redesign. If used at implementation time, these metrics could identify "hot spots" within the code, i.e., areas of probable errors in interfaces. If used at testing or integration points then these metrics could help to plan a strategy for interface testing. Finally, if used at maintenance these measures could help to effect quality maintenance procedures.

Conclusions

We believe that this metric tool can assist software developers in their effort of quality enhancement. If metrics are applied early in the software life cycle, potential problem areas may be detected and corrected early. Correcting problems early within the design and implementation of software systems can result in a great cost savings. By having designers and programmers use this tool to augment their design, coding and testing process, their final software product will be more robust. This tool can make a large contribution to the software development community towards their goal of achieving a quality software product.

InterNet Contact: henry@vtodie.cs.vt.edu

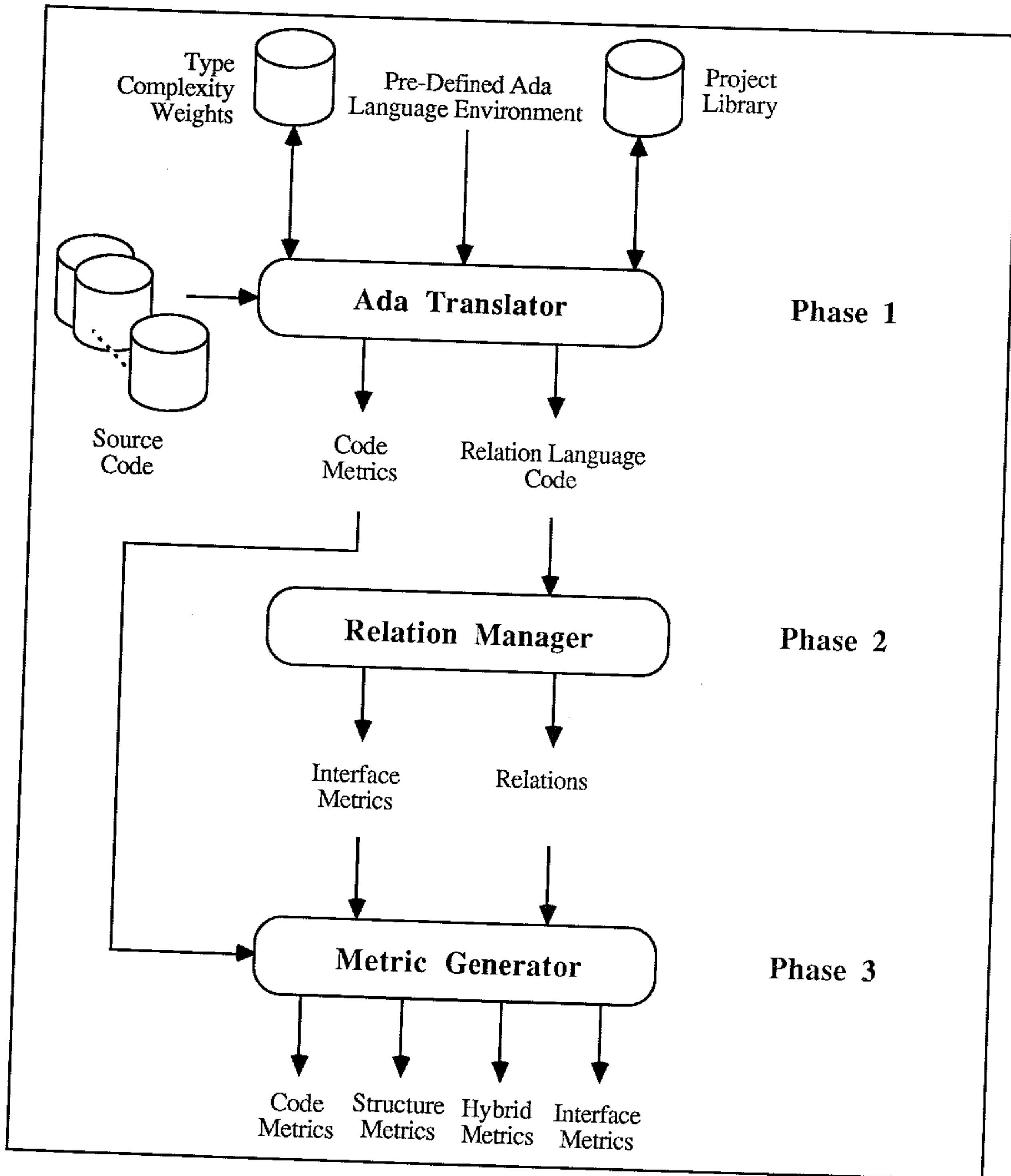

Figure 1. The Software Metric Analyzer

Bibliography

[CHASA82] Chase, Anna I. and Mark S. Gerhardt, "The Case for Full Ada as a Design Language," *Ada Letters*, Vol. 2, No. 3, Nov./Dec. 1982, pp. 51-59.

[CONTS86] Conte, S. D., H. E. Dunsmore, and V. Y. Shen, *Software Engineering Metrics and Models*, Reading, MA: The Benjamin/Cummings Publishing Company, Inc., 1986.

[DEMAT82] DeMarco, Tom, *Controlling Software Projects: Management, Measurement & Estimation*, Englewood Cliffs, NJ: Yourdon Press, 1982.

[GABBE83] Gabber, Eran, "The Middle Way Approach for Ada Based PDL Syntax," *Ada Letters*, Vol. 2, No. 4, Jan./Feb. 1983, pp. 64-67.

[GANNJ86] Gannon, J. D., E. E. Katz, and V. R. Basili, "Metrics for Ada Packages: An Initial Study," *Communications of the ACM*, Vol. 29, No. 7, July 1986, pp. 616-623.

[GILBT77] Gilb, Tom, *Software Metrics*, Cambridge, MA: Winthrop Publishers, Inc., 1977.

[HALSM77] Halstead, Maurice H., *Elements of Software Science*, New York: Elsevier North-Holland, Inc., 1977.

[HENRS81a] Henry, Sallie, Dennis Kafura and Kathy Harris, "On the Relationships Among Three Software Metrics," *Performance Evaluation Review*, Vol. 10, No. 1, Spring 1981, pp. 81-88.

[HENRS81b] Henry, Sallie and Dennis Kafura, "Software Structure Metrics Based on Information Flow," *IEEE Transactions on Software Engineering*, Vol. 7, No. 5, Sept. 1981, pp. 510-518.

[HENRS88] Henry, Sallie, "A Technique for Hiding Proprietary Details While Providing Sufficient Information for Researchers; or, Do You Recognize This Well-Known Algorithm?," *The Journal of Systems and Software*, Vol. 8, No. 1, Jan. 1988, pp. 3-11.

[HENRS89] Henry, Sallie, and Roger Goff, "Complexity Measurement of a Graphical Programming Language," to appear in *Software–Practice and Experience*, 1989.

[HENRS90b] Henry, Sallie, and Calvin Selig, "Design Metrics which Predict Source Code Quality," to appear in *IEEE Software*, 1990.

[HENRS90c] Henry, Sallie, and Roger Goff, "Comparison of a Graphical and a Textual Design Language Using Software Quality Metrics," to appear in *The Journal of Systems and Software*, 1990.

[HOWEB84] Howe, Bob and Jean E. Sammet, "Ada Policy Material from Dallas Meeting in October 1983," *Ada Letters*, Vol. 3, No. 4, Jan./Feb. 1984, pp. 131-136.

[JOHNS75] Johnson, Stephen C., "YACC: Yet Another Compiler-Compiler," *Computing Science Technical Report, No. 32*, Bell Laboratories, Murray Hill, NJ, 1975.

[KAFUD85] Kafura, Dennis and James Canning, "A Validation of Software Metrics Using Many Metrics and Two Resources," *Proceedings: 8th International Conference on Software Engineering*, Aug. 1985, pp. 378-385.

[KAFUD87] Kafura, Dennis and Geereddy R. Reddy, "The Use of Software Complexity Metrics in Software Maintenance," *IEEE Transactions on Software Engineering*, Vol. 13, No. 3, Mar. 1987, pp. 335-343.

[LESKM75] Lesk, M. E. and E. Schmidt, "Lex—A Lexical Analyzer Generator," *Computing Science Technical Report, No. 39*, Bell Laboratories, Murray Hill, NJ, 1975.

[LEWK88] Lew, Ken S., Tharam S. Dillon, and Kevin E. Forward, "Software Complexity and Its Impact on Software Reliability," *IEEE Transactions on Software Engineering*, Vol. 14, No. 11, Nov. 1988, pp. 1645-1655.

[LEWIJ89] Lewis, John, and Sallie Henry, "A Methodology for Integrating Maintainability Using Software Quality Metrics," *Proceedings: IEEE Conference on Software Maintenance*, Oct. 1989, pp. 32-37.

[MAYOK89] Mayo, Kevin A., " Definition and Validation of Interface Complexity Metrics," M.S. Thesis, Department of Computer Science, Virginia Polytechnic Institute and State University, December, 1989.

[MCCAT76] McCabe, Thomas J., "A Complexity Measure," *IEEE Transactions on Software Engineering*, Vol. 2, No. 4, Dec. 1976, pp. 308-320.

[RAMAC85] Ramamoorthy, C. V., et al., "Metrics Guided Methodology," *Proceedings: Computer Software & Applications Conference*, Oct. 1985, pp. 111-120.

[SAMMJ82] Sammet, Jean E., Douglas W. Waugh, and Robert W. Reiter, Jr., "PDL/Ada—A Design Language Based on Ada," *Ada Letters*, Vol. 2, No. 3, Nov./Dec. 1982, pp. 19-31.

[SHATS88] Shatz, Sol M., "Towards Complexity Metrics for Ada Tasking," *IEEE Transactions on Software Engineering*, Vol. 14, No. 8, Aug. 1988, pp. 1122-1127.

[WAKES88] Wake, Steve and Sallie Henry, "A Model Based on Software Quality Factors Which Predicts Maintainability," *Proceedings: Conference on Software Maintenance-1988*, Oct. 1988, pp. 382-387.

[WOODS80] Woodfield, S. N., "Enhanced Effort Estimation by Extending Basic Programming Models to Include Modularity Factors," Ph.D. Dissertation, Computer Science Department, Purdue University, Dec. 1980.

[YOURE89] Yourdon, Ed, "Software Metrics: You Can't Control What You Can't Measure," *American Programmer*, Vol. 2, No. 2, Feb. 1989, pp.3-11.

Bryan L. Chappell received his M.S. in Computer Science from Virginia Tech in 1989. Bryan Chappell's interests include: Operating Systems, Compiler Construction, and Software Engineering. Mr. Chappell is currently working for Software Technology, Inc., in Alexandria, Virginia.

Sallie Henry received her B.S. from the University of Wisconsin-LaCrosse in Mathematics. She received her M.S. and Ph.D. in Computer Science from Iowa State University. While attending Iowa State University, Dr. Henry's research interests were in Programming Languages, Operating Systems, and Software Engineering. After completing her Ph.D. when returned to the University of Wisconsin-LaCrosse as an assistant professor and later an associate professor of Computer Science.

Dr. Sallie Henry is currently on the faculty of the Computer Science Department at Virginia Tech. She has been working in the area of Software Engineering for the past eight years. Her primary research interests are in Software Quality Metrics, Evaluation of Methodologies, Software Testing Methodologies, and Cost Modeling. Her most recent work has been in the validation of software quality metrics, with particular focus on applying metrics during the design phase of the software life cycle. The first of two design studies uses an Ada-like PDL for designing software and the other study incorporates quality metrics with a graphical design language. Dr. Henry's research has been supported by funding from NSF, Naval Surface Surface Weapons Center, Digital Equipment Corporation, IBM, Software Productivity Consortium, Virginia's Center for Inovative Technology, and Xerox Corporation. She is a member of IEEE and ACM.

Kevin A. Mayo received his M.S. in Computer Science from Virginia Tech in 1989. Mr. Mayo's interests include: Software Engineering, Compiler Constructure, Human Factors, and selected Artificial Intelligence Topics. Mr. Mayo is currently working toward his Ph.D. in Computer Science from Virginia Tech.