A Methodology for Integrating Maintainability Using Software Metrics

John Lewis and Sallie Henry

TR 89-2

A Methodology for Integrating Maintainability Using Software Metrics

by

John Lewis

and

Sallie Henry

Computer Science Department
Virginia Tech
Blacksburg, Virginia 24061

(703) 231-7584

Internet: Henry@vtodie.cs.vt.edu

A Methodology for Integrating Maintainability Using Software Metrics

(abstract)

Maintainability must be integrated into software early in the development process. But for practical use, the techniques used must be as unobtrusive to the existing software development process as possible. This paper defines a methodology for integrating maintainability into large-scale software and describes an experiment which implemented the methodology into a major commercial software development environment.

I. Introduction

Software maintenance has been recognized as the single most expensive factor in a software project's life. Estimates place the cost of maintenance from 50 to 70 percent of the total life cycle cost of a software system [HALD88]. A survey of software managers indicates that they realize the crucial role which maintenance plays in their budgets. They also stressed the problem that user demands for enhancements and extensions is often overwhelming. A majority even agreed that software maintenance is more important than new development [LIEB78].

Still, the people who manage, design, and implement software face budget restraints and time deadlines which enforce only the minimum criteria for acceptance, that is that the code "works" for some set of test data. Unfortunately, correct software according to the test plan does *not* guarantee good software in terms of maintenance or other quality factors [MYEG76].

The bottom line is that maintenance is the largest financial drain in a software system's life cycle, and the creation of software must be approached from a perspective that minimizes maintenance costs. The control of software maintenance costs should begin long before the product is delivered to the customer. Unfortunately, the cost benefits of controlling maintenance at the development stage is difficult for software management to realize [MUNJ78].

As the complexity level of a piece of software increases, the code becomes difficult to understand and therefore more likely to contain errors. Code containing errors must be modified, but that maintenance is non-trivial since the code is difficult to comprehend. To compound the problem further, it has been shown that programs cannot be made more maintainable by simply changing their code. Belady and Lehman established that maintenance activities tend to *increase* the level of complexity of the code, so more maintenance is likely to be required [BELL76].

This is the ripple effect of software maintenance which must be controlled at the earliest possible point in the life cycle [YAUS88]. Certainly, using software engineering techniques increase the maintainability of software, but they are difficult to measure and enforce. When deadlines come, good software engineering habits yield to brute force methods which produce code that merely does not violate the test data.

Software metrics, when defined and used correctly, have been shown to be good indicators of software complexity [CONS86]. Metrics are quantitative evaluations of software design or code based on some set of criteria which contribute to the software's complexity. If metric analysis results were required to conform to quality tolerances, the development process would not be compromised.

This research is designed to explore the use of specific implementation techniques and tools to reduce the effort, and therefore the cost, required by the maintenance tasks of large-scale software projects.

Kafura and Reddy established the link between the use of software quality metrics and the efforts of software maintenance [KAFD87]. The study explored medium-sized software systems and used the informed input of experts who were intimately familiar with the system to validate the metrics results. Kafura and Reddy concluded that the software metrics agreed with the general maintenance tasks required, that the metrics identified improper integration of functional enhancements, and that the metrics agreed with the expert, subjective understanding of the system.

Wake and Henry performed a similar study using software quality metrics on a small sample of production code from a major software vendor [WAKS88]. Using a maintenance history describing the modifications made to the code, Wake developed regression equations which successfully predicted the need for maintenance in specific sections of the software.

The research described in this paper was performed in an effort to support the theory that software metrics can be used during multiple phases of the software life cycle to monitor maintainability. This paper describes a methodology for integrating maintainability by using software metrics. A brief overview of the metrics used in this study is given in the next section. The methodology is explained in section III followed by an example of how the methodology has been applied in a commercial software development environment.

II. Software Metrics

Over the past several years, software quality metrics have been shown to be valuable indicators in the quantitative evaluation of software and software designs. For any measurement to be useful, it is important to understand exactly what that measurement means and how to calculate it correctly. This section is an overview of the metrics used in this experiment and a discussion of what kinds of information can be determined from each.

The metrics used in this study were chosen for specific reasons. Of the many software metrics that have been proposed, only those which are totally automatable were considered. The amount of data in this analysis immediately disqualifies any subjective evaluation of the code, or even any objective evaluation that requires individual attention or time-consuming processing.

The metrics discussed here are based on a static analysis of source code. No measurements are made pertaining to the run-time execution of the software. The metrics are discussed here at the procedure level.

Software quality metrics can be subdivided into three areas: code metrics, structure metrics, and hybrid metrics. Each class is now discussed and the metrics from each explained.

Code Metrics

Metrics which deal only with the amount of substance a given procedure contains are called code metrics. The underlying theory is simply that the more items requiring attention that exist in a procedure, the harder that procedure is to understand. A code metric measurement tends to identify the internal quality of a procedure.

Three types of code metrics are analyzed in this study. They are lines of code, Halstead's Software Science indicators, and McCabe's Cyclomatic Complexity [CONS86], [HALM77], [MCCT76].

Researchers disagree on how to count lines of code. In this study, lines of code is calculated as the number of statements not including comment or blank lines. As in Pascal, statements are separated with semicolons in the language analyzed in this study. Therefore, the number of semicolons determines the lines of code metric.

Another set of popular code metrics was defined by Halstead. His measurements are based on the counts of operators and operands within a body of code. The following values are calculated for a procedure:

n1 = number of unique operators

n2 = number of unique operands

N1 = total number of operators

N2 = total number of operands

n = n1 + n2 **Vocabulary Size**

$$N = N1 + N2 \qquad \textbf{Length}$$

Since every entity is included in the counts as either an operator or operand, the sum of the unique elements is called the vocabulary size, and the sum of the total count of elements is considered to be the length, or total number of tokens, of that procedure.

Given these definitions, Halstead then defines program volume, program level, and effort.

Program volume is a measure of size based on the length of the implementation and the size of the vocabulary. The program volume can also be interpreted as the number of mental comparisons needed to generate the program code.

$$V = N \times \log_2(n) \qquad \textbf{Program Volume}$$

Halstead uses an estimator for program level, defined as the level of the code in which an algorithm is implemented. Program level is considered inversely proportional to program difficulty. The lower the level, the more difficult it is to implement the algorithm. Program level is calculated as follows:

$$L = (2 / n1) \times (n2 / N2) \qquad \textbf{Program Level}$$

The final Software Science metric used is Halstead's effort, which attempts to quantify the effort required to generate the implemented code. The effort calculation is the fundamental metric considered when using Halstead's Software Science for complexity analysis. Since the volume of the program dictates the number of mental comparisons needed to implement the program, and the program level is the reciprocal of the difficulty, the effort required is expressed as:

$$E = V / L \qquad \text{Effort}$$

Another code metric used in this study is McCabe's Cyclomatic Complexity, which is defined as a count of the independent logical paths through a procedure. Intuitively, the more logical branches that a procedure contains, the more difficult it becomes to trace all possibilities of actual execution.

Graph theory states that, given a strongly connected graph G with one component, the maximum number of linearly independent circuits V(G), also known as the cyclomatic number, is calculated:

$$V(G) = E - N + 2 \qquad \text{where}$$

$$E = \text{Number of edges in the graph}$$

$$N = \text{Number of nodes in the graph}$$

Therefore, McCabe defines cyclomatic complexity to be the cyclomatic number of a procedure's strongly connected control graph.

Structure Metrics

A second class of metrics, structure metrics, are all based in some way on the overall structure of the system being examined. These quantitative evaluations reflect the bigger picture of the programming system structure.

The structure metrics examined are Henry and Kafura's Information Flow metric and Belady's Cluster metric.

The structure metric developed by Henry and Kafura is discussed in

[HENS81]. The measurement is based on the amount of information which flows in and out of a procedure. Formally, Henry and Kafura define two quantities, fan-in and fan-out:

fan-in: The number of flows of information into a procedure plus the number of global data structures from which a procedure retrieves information.

fan-out: The number of flows of information from a procedure plus the number of global data structures which the procedure updates.

The complexity of procedure p is now defined as follows:

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

The product of fan-in and fan-out is squared because the complexity between system components is non-linear.

Belady discusses the complexity of a software system in relation to how it can be subdivided into logical sections called clusters based on the communication that exists among the individual components [BELL81]. He hypothesizes that understanding interconnected elements is more difficult if their number is large. Furthermore, given the elements, complexity is proportional to the number of connections.

The complexity of an individual cluster of elements j is given as:

$$C_j = N_j \times E_j \qquad \textbf{Cluster Metric}$$

where N_j is the number of nodes within a cluster and E_j is the number of edges between those nodes.

Belady goes on to define the complexity of the entire system as the sum of the individual cluster complexities plus the quantity representing the intercluster communication complexity. That calculation is:

$$C = \Sigma C_j + (N \times E_0)$$

where N is the total number of nodes in the entire system, and E_0 is the number of intercluster edges.

Hybrid Metrics

Code metrics are based on the volume of material that make up a given procedure. Structure metrics take into account the procedure as an entity in the entire system. The concept of a hybrid metric combines both types of evaluations. Any arithmetic combination of code and structure metrics is considered a hybrid metric, although many such combinations do not make intuitive sense.

Woodfield's Review Complexity metric is based on the number of times a component of a system needs to be reviewed to completely understand the system [WOOS80]. Two types of connections between procedures are defined. A control connection exists between a procedure and the procedure which invoked it. A data connection exists between two procedures p and q if there is some variable V which is shared.

Woodfield then defines fan_in of a procedure as the count of all control and data connections for that procedure. The complexity of procedure r is given by:

$$Cr = \text{Effort} \times \sum_{k=2}^{fan_in_i - 1} RC^{k-1}$$

Effort is the code metric defined in Halstead's Software Science. RC is a review constant, given as two-thirds in Woodfield's model. This constant was also suggested by Halstead.

Metric Collection

As stated before, no metric was considered for use in this research which was not automatable. When dealing with large-scale software projects, it is infeasible to address any aspect of the code manually. Therefore, a tool to perform the analysis on the code and calculate the metric values was constructed.

The metric analyzer is actually sectioned into two pieces, one to generate the code metrics and a statistical report, and the second to generate the data from which the structure metrics are calculated. The analyzer pieces are shown in Figure 1.

The code metric analyzer statically evaluates each component of syntactically correct source code and generates the metrics for each procedure. At any point, a statistical report can be generated which gives a breakdown by the LOC, Effort, and Cyclomatic Complexity code metrics, calculating means, standard deviations, minimums and maximums. Also, based on a parameter N to the tool, the report identifies the procedures which have the highest metric values considering the top N percent of the total source analyzed. Furthermore, from that highest N percent, the report indicates which procedures overlapped all three code metrics, or any combination of two code metrics.

This overlap report provides an immediate indicator of where potentially serious problems exist. Using threshold values with varying degrees of confidence, the highest violators of metric tolerances can be identified and investigated.

The data used to calculate the structure and hybrid metrics is obtained from the output of an existing tool in the development environment which will be called the Communication Database. This tool is used to inform a system developer of the proper order in which to rebuild the components of a system, based on the use of specific files by other files. The important aspect of the output of the Communication Database for this research is that it identifies communication lines between modules.

III. The Methodology

Tools are created, metrics established, data validated, and techniques defined. How then does all this combine into a unified whole? A major goal of this study was to define a complete methodology for integrating maintainability into large-scale software.

Another concern came quickly to light as this research began. That was the realization that established companies which design and implement large-scale software products are reluctant to make major changes in their software development process. Many managers view academic research as a supplier of interesting but abstract ideas and will not implement any changes which they view as unnecessarily time-consuming. Often times, immediate deadlines and budget restrictions obscure long-term advantages, even when the techniques presented are proven to enhance productivity.

This research is no exception. Maintenance is an expensive portion of the software life cycle, but it is still viewed by many software developers to be a post-production problem. Managers who are concerned with getting the product out the door are not going to tolerate a methodology which is disruptive to their existing software development process. Therefore, an important characteristic of this methodology is that it be as unobtrusive as possible.

To define such a methodology, it is helpful to examine a fundamental software development technique called iterative enhancement and how it relates to current industry practices.

The concept of iterative enhancement, as defined by Basili, describes a development scheme which inherently supports complexity analysis at the implementation stage [BASV75]. Iterative enhancement is a practical approach to the creation of new software systems. In it, a bare skeleton of the system is fully implemented, then that skeleton is augmented by a particular feature. After that feature is fully tested, the next feature is added, and so on.

Each iteration consists of designing the implementation of a selected task, coding and debugging that task, and analyzing the existing partial implementation at that stage. The analysis also defines what tasks should be added to the project list, in addition to the features of the final project which have yet to be implemented. Note that the idea of iterative enhancement as used here is different from other techniques which implement an almost complete system, then iteratively refine and reorganize until an acceptable design and implementation is achieved.

This technique is rarely used in its purest form. However, a common approach in industry borders on this concept. In the process of creating a new software system, many companies set deadlines for specific internal releases of the software. They also predict a final launch date for the first shipment to the customer. These deadlines might be manipulated as unforeseen problems arise, or the release dates can be held firm and justifications given for any feature not appearing in a particular release that was originally scheduled.

Each internal release is designed with particular goals in mind, which allows software management to assess the progress of the development. Between releases, individual programmers test their code, and when a collection of related modules are ready, integration tests are performed on larger sections of the code. The designated release points allow complete

system tests to be performed.

These release points correspond to the full system analysis stage in the iterative enhancement technique. Therefore, they also provide opportunities to measure the complexity of the system at intermediate points in its development, and determine where complexity problems are being created.

In large-scale system development, the use of the iterative enhancement concept can also be brought down to lower levels in the development structure. Unit managers define similar task lists for the work necessary for their section of the system. Before a system release point occurs, the smaller sub-systems are frozen so that unit testing can be performed to establish confidence in the code which falls under these individual umbrellas.

Each of these points are appropriate places for metric analysis to occur. Errors found at this level are more easily fixed since the scope of where the problem originates is smaller and better understood. Metric evaluation can indicate where that important testing should be concentrated and where error-prone code exists. Figure 2 illustrates the integrated concepts of internal releases and complexity analysis.

Even individual programmers can work on their code with this technique. Once a programmer determines that the code he is responsible for is fully implemented, metric evaluation can pinpoint exact procedures which will probably, if not immediately, cause problems. This identified code can be rewritten or at the very least thoroughly tested. The individual tests run by programmers can uncover many problems that might not show up until further in the testing hierarchy under regular circumstances. Programmers are highly motivated to find any problems in their code before it advances to higher levels of testing where someone else informs them they have a problem.

Complexity analysis can therefore be integrated into extremely early points of the development and testing process. The earlier an error is uncovered and fixed, the less costly the change.

Now we can observe the software development process with the metric evaluation fully integrated into all stages.

Once the functionality of an individual internal release point is defined, the system designed, and the functionality decomposed to unit and programmer levels, the implementation process begins. While studies have shown that metric analysis is useful at the design phase, specific design methodologies must be used [HENS87] [HENS88]. On the other hand, all logic is eventually implemented, which provides a syntactically specific point to do the analysis. The methodology proposed here is based at the implementation and testing levels since this causes the least disruption to the existing development process.

Individual programmers begin implementing their relatively small portion of the logic for the overall system. Once they establish a functional subset of their code, they use the metric analysis tool to generate metric values and a statistical evaluation of their code based on the metrics. This gives immediate feedback as to the volatile portions of their code.

Based on their analysis, portions of their code can be rewritten to reduce the complexity levels. This rewriting process itself is highly likely to uncover logical errors in previously complex procedures. If rewriting is not considered necessary or feasible, the programmer's testing efforts can at least be concentrated on the potentially dangerous code.

Once the programmers have established their portions of the system, pieces are brought together for unit tests. At this stage, the analysis can be run on the larger subset of the overall system. Again, the metric analysis can identify areas that are potentially dangerous, this time from a slightly more

global viewpoint. Structure metrics tend to become more important at this stage since there is more cohesive communication among the larger components of the code.

After unit testing, a full internal release is established. Metric analysis at this stage can give an even more global view of the developing system. Errors concerning integration problems and high-level communication are likely to be uncovered here.

Once the system internal release is tested, new tasks to fix errors and increase the growing functionality of the developing system are defined. Progress toward the next internal release is begun and the cycle repeats itself. Eventually, the last internal release is performed with little or no problems detected, and the product is scheduled for an external release.

This methodology was presented to a commercial software development organization with encouraging results. Plans were made to distribute the metrics tool to various levels of the developing system process, including the individual programmers.

An important caution must be made concerning the use of the metric tool. Any evaluatory technique such as the one described in this research concludes with the result that something is good or bad. Often managers are tempted to use software metrics as yet another way to judge the productivity and usefulness of individuals. This is a dangerous practice. If used as a managerial retribution tool, the entire concept of metric evaluation will be undermined.

The metric tool should be used to evaluate code, not people. Some individuals might be scared into using the tool, but for the most part the use of metrics in that manner will simply cause disharmony and frustration. If the methodology is used as intended, to help guide testing processes and identify error-prone software, the end result will be a software product that is more

reliable and maintainable.

The methodology described in this section incorporates the various techniques and observations established by this research. Since it concentrates on integrating the use of metric analysis without requiring another complete level to the development hierarchy, the use is practical as well as useful.

IV. Application of the Methodology

The methodology described in the previous section was actually applied in a commercial environment. Over 7000 procedures from a given internal release of the project code is analyzed. The software from the project is used to control the functions of a stand-alone machine. Therefore, the system has characteristics of real-time software, in that it must respond almost immediately to user requirements and error conditions. It is a stand-alone system, and therefore contains its own operating system used to control peripherals and the environment. Finally, the system is highly computative in terms of graphics and data management.

Integration and use of the metric values is the next step in this research effort. Error data corresponding to a given release of the software system is used in an attempt to validate the metric results. Statistical correlations of the metrics to each other and to the errors validate the results.

Correlations among the various metrics and between the metrics and the error data are essential to gain an understanding of how useful the metrics are at indicating danger areas and determining which metrics focus on similar attributes.

The correlation matrix for the code metrics is displayed in Figure 3. LOC refers to Lines of Code, N, V and EFF are Halstead's length, volume and effort, CC is McCabe's Cyclomatic Complexity, and ERR represents the number of errors.

Note that all of the code metrics correlated very high (values ranging from 0.852 to 0.998) with each other. This verifies previous study's results and indicates that the code metrics all tend to measure similar aspects of the code, namely the internal complexity [CANJ85] [HENS88].

The correlations between code metrics and defect occurrences (errors) is also significantly high (0.814 through 0.843). This indicates that the code metrics do successfully establish where the errors exist in the source code. The fact that no one metric shows a substantially higher correlation to errors than the others indicates that a single given metric cannot used as "the" metric for the source language and the development environment.

Figure 4 shows the correlations among two code metrics (LOC and V) and the structure and hybrid metrics. WOOD represents Woodfield's Review Complexity, CLUS is Belady's Cluster metric, INFO represents Henry and Kafura's Information Flow metric, and ERR again represents the number of errors.

Error data is once again correlated to all the metrics. Note that the information flow metric does not correlate well with the code metrics. Again, this verifies past studies and indicates that the Information Flow structure metric and code metrics are examining different aspects of software complexity.

The Woodfield hybrid metric correlated well with the code metrics. This is due to the fact that the hybrid metric is, by definition, affected greatly by Halstead's effort code metric. Belady's cluster metric also correlated fairly high to both the code metrics and the other structure metrics. Since this is not a hybrid metric and not inherently affected by code metrics, this would indicate that the cluster metric addresses aspects of both types of complexity.

The correlations to errors are considerably higher for the structure and

hybrid metrics than for the code metrics. This indicates that the structure data yields a higher understanding of error-prone complexity in this environment.

V. Conclusions

The results of this study are both encouraging and educational. The research as a whole was accepted well in the commercial environment it was targeted for, and definite plans are in the works for the fruits of this labor to be permanently implemented. Specifically, the following sum up the results of this research:

- Software quality metrics have again been shown to identify error-prone software due to a high level of complexity.

- Maintainability is a characteristic which can (and should) be integrated into software as it is produced, instead of dealing with it only as a post-production process.

- A methodology is defined to integrate maintainability using software metrics as an evaluation scheme for guiding the testing processes already in place.

- Practical use dictates that such defined methodologies should disrupt as little of the existing development process as possible.

Overall the results were positive and motivating. The concentration on the maintenance aspect of the software development process must be continued in order to reduce the overwhelming cost of software production and service.

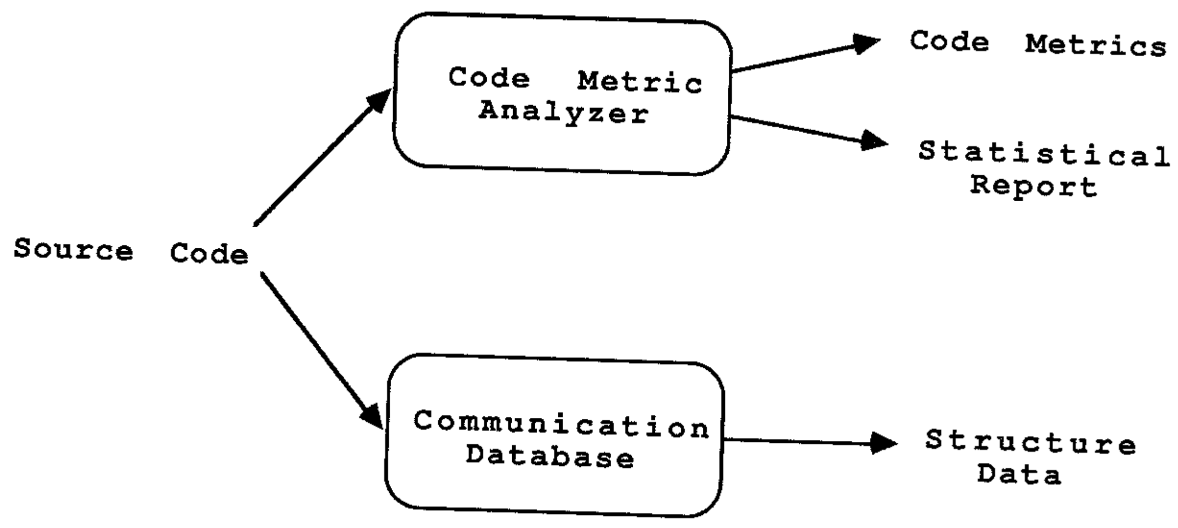

Figure 1. The Metric Analyzer Tool

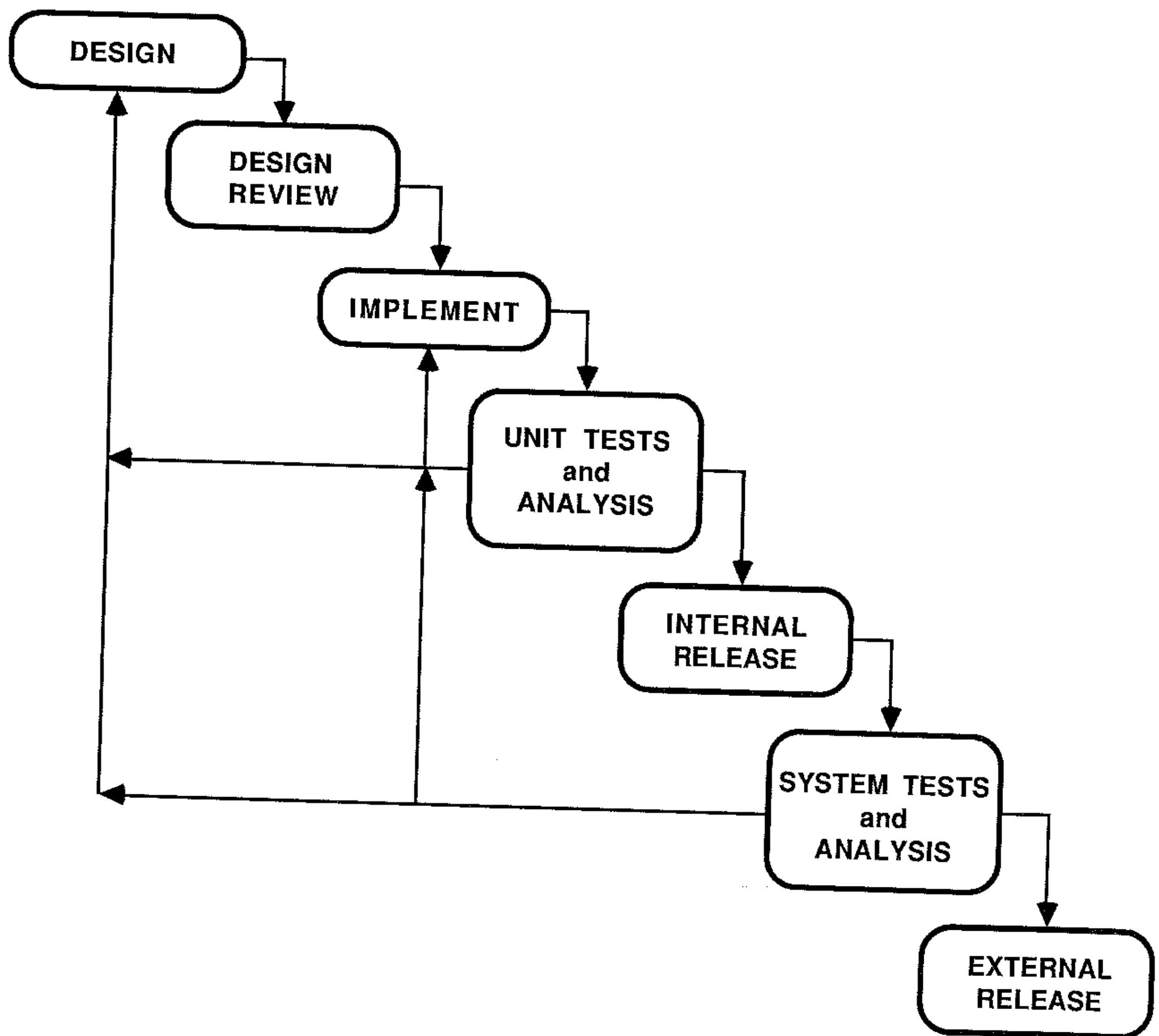

Figure 2. Large-scale development process with release points and metric analysis.

	LOC	N	V	EFF	CC	ERR
LOC	1.00					
N	0.987	1.00				
V	0.982	0.998	1.00			
EFF	0.852	0.884	0.908	1.00		
CC	0.989	0.998	0.993	0.861	1.00	
ERR	0.837	0.831	0.843	0.828	0.814	1.00

Figure 3. Intermetric and Error Correlations for Code Metrics.

	LOC	V	WOOD	CLUS	INFO	ERR
LOC	1.00					
V	0.982	1.00				
WOOD	0.925	0.901	1.00			
CLUS	0.925	0.891	0.999	1.00		
INFO	0.775	0.775	0.930	0.925	1.00	
ERR	0.837	0.843	0.936	0.919	0.884	1.00

Figure 4. Intermetric and Error Correlations including Structure and Hybrid Metrics.

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, 1975, pp. 390-396.

[BELL76] Belady, L.A., Lehman, M., "A model of large program development," IBM System's Journal, No. 3, 1976 pp. 225-252.

[BELL81] Belady, L.A., Evangelisti, C.J., "System Partitioning and Its Measure," Journal of Systems and Software, Vol. 2, 1981, pp. 23-39.

[CANJ85] Canning, J., "The Application of Software Metrics to Large-Scale Systems," Ph.D. dissertation, Department of Computer Science, Virginia Tech, April 1985.

[CASP80] Cashman, P.M., Holt, A.W., "A Communication-Oriented Approach to Structuring the Software Maintenance Environment," ACM SIGSOFT, Software Engineering Notes, Vol. 5, No. 1, January 1980, pp. 4-17.

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

[HALD88] Hale, D.R., Haworth, D.A., "Software Maintenance: A Profile of Past Empirical Research," IEEE Conference on Software Maintenance, November 1988, pp. 236-240.

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

[HENS81] 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.

[HENS87] Henry, S.M., Goff, R., "Complexity Measurement of a Graphical Programming Language," Technical Report TR-87-35, Virginia Tech, November 1987.

[HENS88] Henry, S.M., Selig, C.L., "A Metric Tool for Predicting Source Code Quality from a PDL Design," Proceedings of the Workshop on Software Design Metrics, Melbourne, Florida, March 1988.

[KAFD87] Kafura, D., Reddy, R.R., "The Use of Software Complexity Metrics in Software Maintenance," IEEE Transactions on Software Engineering, Vol. SE-13, No. 3, March 1987, pp. 335-343.

[LIEB78] Lientz, B.P., Swanson, E.B., Tompkins, G.E., "Characteristics of Application Software Maintenance," Communications of the ACM, June 1978, Vol. 21, No. 6, pp. 466-477.

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

[MUNJ78] Munson, J.B., "Software Maintainability: A Practical Concern for Life-Cycle Costs," IEEE Computer Software and Applications Conference, 1978, pp. 54-59.

[MYEG76] Myers, G.J., *Software Reliability, Principles and Practices*, New York, John Wiley & Sons, 1976.

[WAKS88] Wake, S., Henry, S., "A Model Based on Software Quality Factors which Predicts Maintainability," IEEE Conference on Software Maintenance, November, 1988, pp. 382-387.

[WOOS80] Woodfield, S., *Enhanced Effort Estimation by Extending Basic Programming Models to Include Modularity Factors*, Ph. D. Disertation, Computer Science Department, Purdue University, 1980.

[YAUS88] Yau, S.S., Chang, P.S., "A Metric of Modifiability for Software Maintenance," IEEE Conference on Software Maintenance, November 1988. pp. 374-381.