On the Benefits and Difficulties of a Maintainability via Metrics Methodology

By John A. Lewis and Sallie Henry

TR 90-33

On the Benefits and Difficulties of a Maintainability via Metrics Methodology

by

John A. Lewis

and

Sallie M. Henry

Computer Science Department
Virginia Tech
Blacksburg, Virginia 24060

Internet: henry@vtopus.cs.vt.edu

On the Benefits and Difficulties of a Maintainability via Metrics Methodology

(abstract)

A metrics methodology can dramatically reduce the problems associated with software maintenance. However, several issues must be addressed in order to develop and use these techniques successfully. This paper defines a metrics methodology which is designed to deliberately integrate maintainability into software as it is being developed. The benefits of using this approach are discussed. Then several issues which complicate the development and use of the methodology are examined. Previous maintenance studies which incorporate the methodology into two different commercial environments are used to demonstrate the difficulties in implementation and contrast the differences in approach.

1. Introduction

Software maintenance is the largest financial drain in the software life cycle [HALD88]. Many approaches have been offered to control maintenance tasks and reduce the associated costs. These solutions have achieved various levels of success, but no single approach will result in even one order of magnitude improvement [BROF87]. Therefore, the approaches which show promise must be handled in a manner which will maximize their contributions.

The use of software metrics has been successfully applied to the problem of software maintenance [KAFD87]. Methodologies based on metrics can facilitate maintenance tasks, improve the quality of the results, and predict the need for further maintenance efforts [WAKS88] [LEWJ89]. If properly designed and integrated, this methodology can reduce the need for post-production maintenance and facilitate the maintenance tasks (both functional enhancements and defect removal) which are inherent in the development process. However, several issues must be addressed in order to successfully use these techniques.

The basic methodology discussed in this paper is designed for large-scale software production based on some iterative development process. The methodology concentrates on source code analysis because design analysis requires a syntactically specific design language. Many organizations would have to modify their design processes to accommodate the methodology. A major benefit of this methodology is that it is introduced into a new environment with extremely little disruption to the existing development processes.

This paper defines a maintainability via metrics methodology and identifies the benefits of using it. Then, several issues are examined which can cause difficulty in the implementation of the methodology. Two independent studies which introduced the methodology into commercial organizations are used to form the foundation of the discussion of these potential problem areas.

Section 2 defines a maintainability via metrics methodology and discusses research which integrates this approach into two different environments. Section 3 describes the

metrics used in both maintenance studies. The benefits of using a metrics methodology to improve maintainability are discussed in Section 4. Several fundamental difficulties in developing and using the methodology for a given environment are discussed in Section 5. Finally, Section 6 summarizes the relevant points of the paper.

2. Maintainability Using Software Metrics

Before identifying the benefits and difficulties of a maintainability via metrics methodology, the fundamental approach must be understood. This section defines the methodology and describes two maintenance studies which integrate it into two different production environments.

2.1 What is a Maintainability via Metrics Methodology?

The goal of the methodology is to deliberately integrate the characteristic of maintainability into software as it is developed. Concentrating on maintenance tasks after final release is too little, too late. Maintenance is a repetitive process corresponding to the iterative nature of large-scale software production. While the methodology does facilitate post-production maintenance tasks, the main purpose of the approach is to evolve software which is characteristically maintainable at each iteration of the development process.

The methodology uses software complexity metrics to identify high risk areas in the code. Metrics are used to evaluate software according to specific criteria and produce a quantitative measure on a static scale. Therefore, software complexity metrics are used to assess the complexity of a procedure relative to other procedures evaluated in the same manner. Each metric contains an inherent definition of what constitutes complex code. Specific metrics are discussed in Section 3.

Complex code is error-prone and difficult to maintain. In the methodology described in this paper, software complexity metrics are used to identify problem areas within the system. This identification process may involve several types of analysis performed on multiple metrics. Once identified, specific actions are taken to:

- reduce the complexity of the code through further abstraction or reimplementation, and / or

- thoroughly test the high risk areas to uncover as many existing errors as possible.

Therefore, code which will be enhanced and debugged in future iterations of the development process is forced to adhere to certain complexity tolerances. This process will keep the system maintainable as it is developed. Furthermore, the system is less error-prone, resulting in fewer, less difficult maintenance tasks before and after the release of the product. The benefits of the methodology are discussed further in Section 4.

Metric analysis is performed on syntactically correct source code. This analysis can be executed at any (or all) levels of system composition corresponding to testing efforts. Therefore, metric analysis can be performed on individual procedures, integrated units, or full systems. Because of each metric's unique view of complexity, the importance of a given metric may vary depending on the level of analysis.

The metric analysis methodology compliments the iterative process of large-scale software development. Several issues which complicate the integration of the methodology into a new environment are discussed in Section 5. These issues were discovered through two maintenance studies which will be used as examples throughout the remainder of this paper. These studies are described in the following section.

2.2 Two Commercial Examples

Two studies which integrate the maintainability via metrics approach into commercial organizations are presented in [WAKS88] and [LEWJ89]. Production systems under development in both environments were analyzed to verify the methodology and refine analysis techniques. These studies will be used as examples in our discussions on the benefits and difficulties of this approach.

In both studies, the organizations into which the methodology was introduced are state-of-the-art, large-scale software producers. Unfortunately, both environments deal with highly proprietary information which restricts the discussions of the systems used to develop the methodology and even prohibits revealing the names of the corporations. Handling proprietary information is an important issue and is discussed in Section 5.6.

The first study examined 193 procedures containing approximately 15,000 lines of C code including comments and blank lines. The system analyzed is version 2.0 of an actual product. A code library is used to monitor changes made to the code. This modification data is used in an analysis technique to predict the need for future maintenance efforts.

The second maintenance study performed analysis on over 6000 procedures of source code written in a language similar to Ada. The software is used to control the functions of a stand-alone machine. The system contains operating system code, real-time applications, graphics, and peripheral control routines. Several analysis techniques were developed in this study.

3. Software Complexity Metrics

This section describes all of the software complexity metrics used in the studies identified in Section 2.2 . The metrics described are:

- Lines of Code,
- Halstead's Software Science Indicators,
- McCabe's Cyclomatic Complexity,
- Henry and Kafura's Information Flow,
- Belady's Cluster Metric, and
- Woodfield's Review Complexity.

The reader is encouraged to skip this section if familiar with the metrics. However, a thorough understanding of the metrics and their underlying conceptual approach is needed for the discussions which follow.

Software complexity metrics can be divided into the following categories:

- Code metrics,
- Structure metrics, and
- Hybrid metrics.

The following sections separate the metrics into these three categories and define the aspects of complexity each attempts to measure.

3.1 Code Metrics

Code metrics examine the internal complexity of a procedure by analyzing the amount of information within the procedure or assessing the logical complexity of the code. The code metrics discussed are Lines of Code, Halstead's Software Science [HALM77], and McCabe's Cyclomatic Complexity [MCCT76].

3.1.1 Lines of Code

One popular metric is the measure of how many lines of code (LOC) exist in a given procedure. The intuition is that a procedure with twenty LOC, for example, is considerably less complex than one with fifty LOC. Most software engineers agree that the overriding consideration is that a procedure should perform a single logical function. This can usually be accomplished in less than fifty lines of source code [MYEG76].

The LOC metric is a simplistic view of software, but can be used to identify flagrant violators of complexity tolerances. When analyzing a software system whose average LOC per procedure is thirty (an established norm), a particular procedure measuring one hundred lines of code instantly warrants attention. That procedure should be extremely straightforward at first glance, and even then most software engineers would suggest further abstraction.

Researchers have not agreed on how to count a line of code, though most agree that the measurement should not include blank lines or comments. They also debate whether variable, type, and constant declarations should be considered [CONS86]. It is usually sufficient to choose a logical method and be consistent with its use.

3.1.2 Halstead's Software Science

Another set of popular code metrics is defined by Halstead [HALM77]. His measurements are based on the counts of operators and operands within a procedure of code. The following four initial values are calculated:

- n1 = number of unique operators
- n2 = number of unique operands
- N1 = total number of operators
- N2 = total number of operands.

To calculate these values, everything in the source code is considered to be either an operator or an operand. Operands consist of any entity being operated upon, such as constants and variables. Operators are all other tokens. These include arithmetic and boolean operators, control statements, and assignment structures. Distinct keywords or symbols which cooperate to perform one function are often logically grouped and considered to be one operator, such as the If-Then, While-Do, and Begin-End constructs.

Once these initial counts are taken, the following values are defined:

- Vocabulary Size: $n = n1 + n2$
- Length: $N = N1 + N2$
- Program Volume: $V = N * \log_2(n)$
- Program Level: $L = (2 / n1) * (n2 / N2)$
- Language Level: $\lambda = L^2 * V$
- Effort: $E = V / L.$

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.

Program volume is a measure of size based on the length of the implementation and the size of the vocabulary. Given n distinct elements, the smallest binary representation of a particular element requires $\log_2(n)$ bits. The program volume can also be interpreted as the number of mental comparisons needed to generate the program code. This view

assumes that a programmer selects an element by making a mental binary search through the program vocabulary. This interpretation is important in the derivation of the formula for effort.

Program level is an estimate of the level at which an algorithm is implemented. It is considered inversely proportional to program difficulty. Therefore, the lower the level, the more difficult it is to implement the algorithm. The language level computation is a quantitative measure used to compare two different languages and is independent of the algorithm implemented.

The final Software Science metric defined above is effort, which attempts to quantify the effort required to generate the implemented code. 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 the ratio of these measures.

3.1.2 McCabe's Cyclomatic Complexity

McCabe's cyclomatic complexity is defined as a count of the independent logical paths through a procedure [MCCT76]. Intuitively, the more logical branches that a procedure contains, the more difficult it becomes to trace all execution possibilities.

Based in graph theory, the complexity measurement is taken from a representation of a procedure as a strongly connected graph. Each node in the graph represents a sequential block of code, and each edge represents a logical branching point through the procedure. The graph must have unique entry and exit points.

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:

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

where E and N are the number of edges and nodes in the graph, respectively. Therefore, McCabe defines cyclomatic complexity to be the cyclomatic number of a procedure's strongly connected control graph.

An alternative method of calculating cyclomatic complexity is used if the analyzed code is given in structured form. In this case the cyclomatic complexity is equal to a count of the number of decision points in a procedure plus one. In the graph representation, a decision point is any node which has more than one directed edge from it. This method is an efficient calculation of the metric based on the source language constructs.

3.2 Structure Metrics

Structure metrics examine the relationship between a section of code and the rest of the system. These metrics are based on some evaluation of the communication (via data structures, parameters, etc.) from one procedure to another. Changes and errors in a particular section of code can have dramatic repercussions throughout the system due to these communication lines. Therefore, structure metrics take an overall view of the system structure and represent the magnitude of the potential repercussions. The structure metrics discussed are Henry and Kafura's Information Flow [HENS81], and Belady's Cluster Metric [BELL81].

3.2.1 Henry and Kafura's Information Flow

The structure metric developed by Henry and Kafura is based on the amount of information which flows into and out of a procedure. Two measures are defined:

- 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 these values, structure data is generated representing the flow of information within a routine from input and output parameters, global data structures, and return values from functions. A flow graph is constructed representing all possible paths through a program which result in updates of global data structures.

The complexity of procedure p is defined as follows:

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

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

3.2.2 Belady's Cluster Metric

Belady examines the complexity of a software system in terms of how well 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 cluster metric calculation is based on a graphical representation of the system, where nodes of an undirected graph represent individual elements of the system, and edges represent a communication line between the elements.

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

- $C_j = N_j * E_j$

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

Belady defines 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:

$$\bullet \quad C = \Sigma C_j + (N * E_0)$$

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

3.3 Hybrid Metrics

Hybrid metrics combine an internal view of a procedure (code metrics) with a measure of the communication connections between that code and the rest of the system (structure metrics). Any linear combination of code and structure metrics can be considered a hybrid metric, although many such combinations do not make intuitive sense. Woodfield's Review Complexity is a hybrid metric [WOOS80]. Henry and Kafura's Information Flow can also be used as a hybrid metric when combined with various code metrics [HENS81].

3.3.1 Woodfield's Review Complexity

Woodfield's Review Complexity metric is based on the number of times a component of a system needs to be examined (or reviewed) in order 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 such that the following conditions hold:

- The variable V is modified in p and referenced in q.
- There exists at least one data path set D(pq), between p and q, such that the same variable V is not referenced in any d_i where d_i is a member of D(pq).

A data path set D(pq) is defined as an ordered set of one or more procedures $\{d_1, d_2, d_3, ..., d_n\}$ such that one of the following conditions hold:

- q calls d_1; d_1 calls d_2; ... ; d_{n-1} calls d_n; d_n calls p.

- p calls d_1; d_1 calls d_2; ... ; d_{n-1} calls d_n; d_n calls q.

- There exists some d_i in D(pq) such that d_i calls both d_{i-1} and d_{i+1}; d_{i-1} calls d_{i-2}; d_{i-1} calls d_{i-3}; ... ; d_2 calls d_1; d_1 calls q; and also d_{i+1} calls d_{i+2}; d_{i+2} calls d_{i+3}; ... ; d_{n-1} calls d_n; d_n calls p.

Woodfield then defines the fan-in of a procedure as the count of all control and data connections for that procedure. This definition of fan-in should not be confused with Henry and Kafura's use of the term (Section 3.2.1). The complexity of procedure i is given by:

$$\bullet \quad C_i = \text{effort} * \sum_{k=2}^{fan\_in_i - 1} RC^{k-1}$$

where effort is the code metric defined in Halstead's Software Science [HALM77], and RC is a review constant, given as two-thirds in Woodfield's model. This constant was also suggested by Halstead.

3.3.2 Information Flow as a Hybrid Metric

Any code metric can be used to weight Henry and Kafura's Information Flow metric. For example, using the Lines of Code (LOC) value to weight the Information Flow metric yields the following hybrid definition of the complexity of a procedure p:

$$\bullet \quad Cp = LOC * (\text{fan-in} * \text{fan-out})^2$$

LOC is defined in Section 3.1.1 and fan-in and fan-out are defined in Section 3.2.1. Any code metric could replace the lines of code count in the equation above to yield another hybrid version of the Information Flow metric.

4. Benefits

Both studies described in Section 2.2 successfully demonstrate several worthwhile objectives in the area of software maintenance. This section describes the benefits of using a maintainability via metrics methodology, namely:

- Iterative, consistent improvement of the evolving product,
- Ability to adapt the methodology to multiple domains, and
- Low impact integration.

These issues are addressed individually in the following sections.

4.1 Product Evolution

One benefit of using the methodology is that it quietly modifies the fundamental attitude of state-of-the-practice software development. Today's standard approach is often multiple iterations of: design, code, test, fix errors. The result is a product which may work according to test results, but is fundamentally unstable due to the haphazard maintenance tasks which were a large part of its development. Using a metric methodology, the maintenance tasks are guided by complexity considerations, which are controlled at each iteration, therefore resulting in a stable product which gains strength as it evolves. The overall appearance of the development process is not that different, but the fundamental results are quite different.

The methodology is designed to guide the various development activities that are inherent in creating a new software product. Metric values determine which code segments are error-prone and therefore deserve further attention. The specific actions taken depend on the nature of the problem uncovered, but range from concentrating testing efforts on the problem areas to complete reimplemenation or redesign to reduce complexity.

There are advantages and disadvantages to various actions taken as a result of metric analysis. A major validation problem is that it is often infeasible to exhaustively test new code. At the very least, the metrics methodology can guide testing efforts to the areas in the system where defects are likely to exist. Early detection of these errors is a fundamental goal of the development process. Furthermore, the high complexity of the code makes errors more difficult to find and correct, making early detection even more important.

Concentrating testing efforts helps the situation, but is not much of an improvement over the classical method which results in "patched" code. A more beneficial use of the methodology is to modify the implementation (or design if necessary) in order to reduce the complexity of the code. It will then be less likely to contain errors, and the errors which do exist will be much easier to find and correct. Maintenance tasks make a system less maintainable and often introduce new errors. With the metrics methodology, the maintenance activities are reduced and the maintenance ripple effect does not propagate through the system.

Some code segments, for one reason or another, are inherently complex. Algorithms designed for extreme efficiency, for example, often contain code fragments which are not intuitive and the metrics analysis may identify these areas as potentially dangerous. In these cases, the efficiency is the dominating consideration, and no changes to the code should be made. However, the problem is still brought to the forefront and other possibilities (extended documentation, testing, etc.) can be used to raise the level of confidence in current and future maintenance activities.

No matter what specific actions are taken as a result of the metric analysis, the important point is that potential problems due to complexity are identified early in the development process. The metric methodology provides simple but important techniques to guide the development processes.

4.2 Adaptability

A second benefit of using a maintainability via metrics methodology is that it can easily be augmented or adapted to include areas of concern besides code complexity. For example, style metrics can be used to identify acceptable documentation or format considerations [ARTJ89]. Other metrics can be defined to concentrate on almost any statically discernable characteristic of the code, allowing the user to tailor it to particular elements of interest.

Furthermore, the metrics defined in Section 3 represent the ones used in [WAKS88] and [LEWJ89]. Many complexity metrics have been defined in the literature and might be substituted for the ones presented in this paper. The metrics used in these maintenance studies were chosen for their successful results in the past and because they are completely

automatable. Since the methodology is geared toward large-scale software development, it is infeasible to use metrics which require extensive human interaction or processing effort. Section 5.1 discusses evaluating an individual environment and language to help determine which metrics to include.

Since the metric value collection is automated, software must be developed for individual metrics. Therefore, some consideration should be given to changing or adding metrics to the methodology set. However, the general processes and integration scheme of the methodology are such that changes can be made without any interruption to the current analysis. The low-impact integration of the methodology is another important benefit and is discussed in the next section.

4.3 Integration

Yet another major benefit of the methodology is that it is introduced into a new environment with very little impact on the existing development practices. Assuming an iterative development scheme for large-scale software products, the methodology is designed to compliment the existing processes, not change them.

The integration of the methodology begins by identifying the testing phases through which the product is subjected. For this discussion, we will address three general levels of testing:

- Unit testing,
- Integration testing, and
- System testing.

In the methodology, metric analysis corresponds to testing efforts, therefore no additional efforts are needed to secure a current version of the software. Also, depending on the level at which the analysis is performed, some metrics may be more relevant than others.

Unit testing is performed by an programmer on his (or preferably another's) individual code sections. Often these code sections do not stand alone and must have driver and stub routines for test execution. At the unit test point, a programmer can perform metric analysis and concentrate on problems in detail. The analyzed code is small

in quantity and the analyzer often knows the details of the source code intimately. Code metrics are more informative at this point because little of the supporting structure is present to give full meaning to the structure metrics.

Integration testing involves some combination of individual units in their proper relationships. The entire system structure is not complete, but stand alone subsections are present and functional. Therefore, code, structure, and hybrid metrics can offer invaluable insight to the complexities of the source code in the integrated unit and can be viewed from both global and detailed perspectives. A single individual, perhaps a quality control or testing specialist, can perform the analysis at this stage.

System testing is usually performed at a formal internal release point during which all system code is frozen and integrated into a complete system. Depending on the size of the system, the best perspective at this point is a global view concentrating on the structure and hybrid metrics. Once again, a single individual can perform the analysis, possibly consulting others during the interpretation portion of the analysis.

Using these techniques, the methodology can be integrated into the existing production scheme at various levels with little disruption. However, the integration process is discussed further in Section 5.2, exploring the difficulties that can be introduced if the methodology is not integrated unobtrusively.

5. Difficulties

The benefits of the maintainability via metrics methodology are substantial. However, care must be taken to correctly integrate the methodology into a new environment. The following are issues which can cause problems if not handled properly:

- Initial evaluation of the language, environment, and development paradigm,
- Integration of the methodology into an existing development structure,
- Development of metric analysis techniques,
- Development of automated tools for metric collection and analysis,
- Use of historical data,
- Handling proprietary information,
- Interpretation of metric values, and

- Proper use of the methodology.

These issues are discussed in detail in the following sections.

5.1 Initial Evaluation

If a random set of metrics is chosen to analyze source code of a language, without regard to the particulars of the language or the environment, the methodology is not likely succeed from either a global or detailed perspective. From a global point of view, the methodology will not make the best use of the existing environment and development life-cycle. From a detailed perspective, the metrics used may not be optimal for assessing the particular complexity problems of the language analyzed.

Three distinct aspects of a new organization must be critically evaluated before developing a metrics methodology:

- The programming language,
- The development environment, and
- The software development paradigm.

Each of these issues may affect the metrics used in the methodology and the particulars of the integration scheme.

The programming language is naturally an important aspect to consider when deciding what metrics to use. Most complexity metrics in the literature are defined generically and can be used for many languages. However, there are two reasons why new metrics might be defined or published metrics modified for a particular language. First, if the language in question is not universally used (an in-house or new language), new constructs or techniques may warrant special or careful treatment. Second, if there have been specific problems in the past (with a standard or non-standard language), the metrics chosen or defined might concentrate on the specific aspects of the problem. For example, in the in-house language used in [LEWJ89], there exists a particular construct which is unnecessarily complex and can be avoided in most cases. A simple count of this construct was added to the metric set for that language to catch unnecessary use of the construct.

The development environment is often more tailored to a particular organization than the language used. Therefore, metrics might be used which compliment the various combination of hardware and software. Also, integration of software tools such as compilers, debuggers, browsers, etc. may suggest where metric analysis might be unobtrusively accomplished and perhaps completely integrated into the development plan.

The software development paradigm, or life cycle, defines the stages through which the evolving software product steps as it is developed. This process must be examined to determine the appropriate corrective actions which will be taken once error-prone code is identified. The development process also must be examined to determine the appropriate points at which to perform the metric analysis. The best analysis points probably correspond to the various testing levels which exist in the process, but these points must be tailored to the specific software life cycle used. The problems associated with integration are discussed further in the next section.

5.2 Integration

As discussed in Section 4.3, the methodology can be integrated into the existing development scheme with little disruption. Not only is this a major advantage of the methodology, it is, in most situations, a necessity. Therefore this section explores the problems which can be faced by trying to introduce the methodology in a conspicuous manner.

Most organizations have an established, structured development strategy with well-defined stages through which the evolving software product progresses. For large-scale production, this process usually uses iterative, repeating stages as new functionality is introduced and corrective actions are taken. This established process will resist change unless faced with monumental problems.

While the benefits of using the methodology are substantial, the direct results are often intangible. A reduction in error rates and maintenance efforts are high-level benefits resulting from daily efforts. If the daily efforts seem oppressive or detracting, the "big picture" can be ignored in favor of deadlines and arbitrary budgets, which in the long run is counterproductive.

Considering a new methodology is often contingent on the amount of interruption it will introduce into the daily lives of the developers. Therefore, from the standpoint of practical use, the methodology must be unobtrusive. Using the techniques described in Section 4.3, the metric analysis corresponds to testing efforts and therefore does not require unusual preparation. The analysis process itself is automated and is primarily effortless. The only substantial impact is the execution of corrective actions when error-prone code is identified. As discussed in Section 5.2, these actions can be as involved as complete reimplementation or as small as concentrating testing efforts on that section, depending on the nature of the problem.

5.3 Analysis Techniques

This section describes some analysis techniques that have proved useful in interpreting and acting on the quantitative metric values. These techniques identify error-prone code by determining which sections of code are most complex based on a collaboration of metric values. Care must be taken to assure that these techniques are used intelligently and without bias. The techniques examined are:

- Fundamental statistics,
- Threshold analysis, and
- Prediction equations.

Fundamental statistics can be used to get a global picture of the metric analysis results. Sample means and standard deviations are easily computed, but must be carefully interpreted. For example, the average metric value for a set of code is not necessarily an acceptable (safe) value. Another simple procedure cross references code segments with the highest metric values (perhaps the highest 10%) across several metrics. If a code segment appear in all, it may warrant further attention.

The threshold analysis technique involves separating the metric values into categories depending on the level of concern raised by that metric. This technique requires setting tolerances on the metric values. Figure 1 shows three code metrics separated into three categories: a "safe" zone, the range where a flag should be raised, and the level past which an alarm should be sounded. These tolerances are an initial estimate used in [LEWJ89].

For a particular organization (language, environment, and paradigm), general rules should be set and followed to determine which code segments require further investigation. For example, using the situation in Figure 1, metric values which raise two flags, one alarm and a flag, or anything worse should be investigated. Of course, in a given methodology, as many metrics as possible should be used in threshold analysis and the guidelines justifying further effort should be adjusted accordingly. Additional categories can be used, but given that these are general complexity metrics and the tolerance levels are somewhat arbitrary, it is unwise to get too specific.

Threshold analysis requires a cooperative effort of the metrics to identify dangerous code. There may be valid reasons why a single metric may exceed tolerance limits, while others indicate that it is not a problem. Metric interpretation is discussed further in Section 5.7.

A third analysis technique is the use of prediction equations. Using multiple linear regression, equations can be generated that predict, from the metric values, the number of errors, lines of code with defects, etc., which can be found within a section of code. The development of these equations is a straightforward statistical process, but requires historical error data for a substantial amount of code, along with the metric values for that code. The following is an example of a predictor equation used in [WAKS88]:

$$\begin{aligned} NLC = {} & 1.27935618 + 0.05500043\,L - 0.001333387\,V + 0.000054797\,E - \\ & 0.11960695\,V(G) - 0.000000142938\,INFO\text{-}E \end{aligned}$$

This equation predicts the number of lines of code that must be changed to correct defects. The metric values used as independent variables are Halstead's length (L), volume (V), effort (E), McCabe's cyclomatic complexity (V(G)), and Henry and Kafura's hybrid information flow metric weighted by effort (INFO-E). Many such equations can be generated, with different combinations of metrics and coefficients. Various statistics can be used to determine which equations best predict the dependent variable (R^2, PRESS, MSE, C(P)).

Once a regression equation is generated, new code can be analyzed using the equation to predict the number of defects in the code. This evaluation can be used to determine which sections of the newly developed system need specific attention.

In both threshold and prediction analysis, the data used to define the techniques ultimately determines their usefulness. These techniques should be refined as more data is made available and as further examination of the language and environment is performed. For example, an educated guess can be made to determine the threshold boundaries initially, but as metric analysis is compared to actual defect rate, these boundaries should be adjusted to accommodate the situation as appropriate. Therefore, the prediction equation and tolerance levels shown above and in Figure 1 should not be arbitrarily used in another organization. They were tailored to particular environments and languages. The techniques used to generate them, however, are very useful.

Historical project data is obviously essential to the development of some analysis techniques. Problems with the collection and accuracy of historical data is discussed in Section 5.5.

Some of these techniques can and should be automated to speed the metric analysis. The following section discusses software tools which collect the metric values and perform interpretation analysis.

5.4 Tool Development

For large-scale software development, it is impractical to consider performing any metric analysis by hand. At the very minimum, a software tool is needed to parse the source code and generate the metric values.

In the first maintenance study described in Section 2.2, the metric collection tool works in three phases. Phase one generates the code metrics and translates the source into an intermediate representation (called relation language) which retains only enough information to generate the structure metrics [HENS88]. Phase two parses the relation language into a series of data entries called relations which incorporate all information flows within the subset of code being processed. The third phase generates the structure metrics from the relations and generates the hybrid metrics using both the code and

structure metrics. The intermediate relation language form is discussed further in Section 5.6.

The second maintenance study used a slightly different approach due to the nature of the data available. The code metrics are generated from an initial parse of the source code, but the code is not translated into an intermediate form. Since structure data for the system was already available in a machine-readable form (called here the communication database), the structure metrics were calculated straight from that information. Hybrid metrics were then generated using both the code and structure metrics. Figure 2 depicts both metric collection tools used in the example maintenance studies.

Development of a metric collection tool requires a substantial time investment initially, depending on the language analyzed and the environment. Once developed, the tool can quickly generate the metrics for hundreds of procedures of source code. The time impact for metric generation depends on the system analyzed and the metrics generated. Some metrics are calculated in a straight-forward manner, while others, in order to fulfill the correct definition, require more extensive processing.

Other tools can be implemented to assist in the interpretation of the metric values. As discussed in the previous section, metric analysis can involve statistics, threshold categorization, and prediction equations. The metric values from the collection tool can be automatically passed into analysis tools for further processing.

The first maintenance study from Section 2.2 did not automate the analysis processes. The second study added a statistical package which computed sample mean and standard deviation, performed cross reference analysis, and sorted the procedures by metric values to identify the highest contributors for each metric (see Figure 2). No automation of threshold or prediction analysis was performed.

Ultimately, a human must examine any error-prone code and determine what action must be taken. Therefore, this process must be performed by a person who understands the theory behind the metrics. Blind acceptance of the metric values is unwise. This problem is discussed further in Section 5.7.

5.5 Historical Data

Many metric analysis techniques are developed using historical project data. For example, the prediction equations described in Section 5.3 rely heavily on some form of error data. By using previous error rates and the metric values for the code containing those errors, equations are formed which can estimate how many defects can be expected in new code, based on the new code's metric values.

While the metric collection can be performed long after the source code is developed, the error data collection must occur as the defects are discovered. Recording this information is relatively cheap and the potential benefits are quite large. Even if not used in a metric analysis process, the error data can shed light on the development process as a whole, indicating the types, locations, and proliferation of errors.

The equation in Section 5.3 predicts the number of lines of code that must be modified for error defect removal. Equations can just as easily be generated to predict the number of actual errors, time impact of errors, or any other quantitative measure that makes sense, as long as the historical data is present from which to develop the equations.

The quality of the data is equally important. For example, the second maintenance experiment from Section 2.2 also demonstrated the ability to predict errors and explored the techniques to generate the equations, but the actual equations which were developed are practically useless. The error data available was collected such that the defects could only be traced back to a large subsystem of the source code. Therefore, the equations predicted the number of errors in an entire subsystem of code, which was of little assistance in new system development. Since then, the organization is collecting error data at lower levels and generating more useful equations.

Historical data collection is essential to develop and refine many metric analysis techniques. The more detailed and accurate the data is, the more robust the analysis techniques can be. Early attention to this process is extremely important.

5.6 Proprietary Information

Many organizations deal with information which is considered proprietary due to financial or security considerations. This fact often deters these organizations from

pursuing independent research. Investigating software development possibilities in commercial environments can lead to significant breakthroughs which may elude purely academic experimentation. Furthermore, avoiding these research efforts is often unnecessary and counterproductive.

Both organizations in the maintenance studies from Section 2.2 deal with highly proprietary information. However, the studies extensively analyzed production systems without the need to disclose any sensitive data. This is accomplished by a careful examination of the information needed to perform the metric analysis.

As discussed in Sections 2.1 and 5.4, metric data is collected by an automated parse of syntactically correct source code. Code metrics are obtained immediately from this initial phase. Structure data is also gathered in this phase from which the structure metrics are calculated. However, the information necessary to generate the structure metrics is independent of the semantic content. Therefore the data can be represented in a form which removes all specific references to what the code accomplishes and leaves only enough detail to generate the structure metrics. Furthermore, this intermediate form can be automatically fed to subsequent phases and then deleted, reducing human intervention.

The first maintenance study used an intermediate form for the structure data called relation language, which is described in detail in [HENS88]. Figure 3 shows a procedure of Pascal source code and its relation language translation. Note that only the fundamental logical structure remains of the source code. There is virtually no possibility of gaining any useful information about the original code, yet enough detail is retained to compute the structure metrics.

The relation language translation replaces all conditional and looping constructs (if, while, repeat, etc.) with the keyword COND. All operators are replaced with the generic operator ampersand (&) and assignments use the colon-equal symbol (:=). To further disguise the source, all identifiers are translated into consistent but meaningless strings, and all constants are replaced by the single constant 100.

The second maintenance study used a similar intermediate representation. The details of the translation vary somewhat due to the nature of the data represented. The parsing tool

from this study used a complex encoding scheme to disguise identifiers. The translation process uses a software key which can be changed to rearrange the encoding scheme.

The generation of the metric collection tool requires knowledge of the source code grammar, but can be tested using non-proprietary code. Once developed and tested, the tool can parse any syntactically correct source code to produce code metrics and structure data. The structure data is then parsed to produce structure and hybrid metrics. Since all proprietary information is removed in the first phase, structure metric generation and metric interpretation can be performed off-site and by uncleared personnel, if desired.

5.7 Metric Interpretation

Metrics quantify software complexity. This provides an established scale on which to compare code segments and gives software developers a means to rank and address error-prone software as it is developed.

However, it is dangerous to accept a metric value with blind faith. Each metric inherently has a definition of complexity which it attempts to quantify. Often these definitions are quite different. To say a routine is error-prone simply because a certain metric has a high value, without understanding what the metric attempts to measure (at least in general terms), is using the methodology incorrectly.

There are valid reasons why a single metric may exceed established tolerance limits, yet still not be considered a problem. This is why the analysis techniques discussed in Section 5.3 use multiple metrics to determine where maintenance efforts need to be concentrated.

Consider, for example, a report routine which performed a large amount of straightforward output. The threshold analysis technique in Section 5.3 might raise a flag from the lines of code metric, but Halstead's effort and McCabe's cyclomatic complexity would be negligible. Therefore, the technique is designed such that some consensus between the metrics must be established before concern is raised.

Furthermore, note that the "safe" category in Figure 1 is in quotes. Certainly, no one should assume, simply because a code section has low metric values, that it has no errors

whatsoever. The point is that errors which do exist will be relatively easy to find and correct because the complexity level of the source is low.

Both maintenance studies in Section 2.2 determined that the code metrics are highly correlated and the structure metrics are highly correlated, but do not correlate with each other. This is because they attempt to measure different aspects of complexity. This may also lead software developers to believe that they only need one code metric and one structure metric (or simply one hybrid metric) to answer their complexity questions. However, this is a dangerous assumption.

Yes, the metrics correlate in the long run. However, as established in Section 5.4, determining what action should be taken in a particular case must ultimately be a human process. In each particular case, two metrics which correlate in the long run may be dramatically different. Furthermore, different metric types may be more informative at different levels of analysis, depending on the amount of code analyzed.

A metrics methodology must make use of as many different metrics as is feasible in order to increase the chances of identifying error-prone code. The underlying concepts behind the metric values must be understood by the analyzers so that appropriate action can be taken in any particular case.

5.8 Using the Methodology

An important aspect of the methodology is the manner in which it is presented to programmers. The purpose of the methodology is to consistently develop a maintainable system by identifying code which deserves further attention due to its complexity. The attitude of both managers and programmers must reflect this purpose.

Management must avoid the tendency to use the metric analysis to determine the quality of an individual rather than the quality of the source code. As discussed in Section 5.7, in some situations it may be appropriate to have code which violates complexity tolerances. Code complexity is a result of the required functionality as well as the techniques used to write the code. Therefore, to assess the quality of a person by the metric analysis of his code is both unwarranted and dangerous.

If managers use metric analysis as a basis for evaluating individuals, programmers will fear the tool and reject the proper applications. Consequently, the entire purpose of the methodology will be undermined. However, if presented correctly, programmers will view the tool and methodology for what it is, a quality control process directed at the product.

One method to assure that programmers do not fear the analysis tool is to give them first access to it, as discussed in Section 5.2. Then before an individual's code leaves his desk, he can evaluate it himself using the metric analysis techniques and determine if any action should be taken. Problem areas which are identified early are also less expensive to correct than those discovered later.

6. Summary

A maintenance via metrics methodology can substantially decrease maintenance efforts during production and after release. Furthermore, the methodology is designed such that the new system gains strength as it evolves as opposed to systematically weakening due to error patches. However, attention must be paid to several issues which can cause problems if not handled properly. Environment differences, integration techniques, use of data, and managerial attitude must be considered carefully in order to maximize the benefits of the methodology.

From a comparison of Sections 4 and 5, it may appear that the difficulties inherent in the methodology outweigh the benefits, but this is not the case. While the benefits are succinct and straightforward, they are also crucial to a successful software development effort. Likewise, the difficulties are presented so that they can be adequately addressed in a timely fashion, not to discourage the use of the methodology.

A maintenance metrics methodology is both useful and practical. Further studies which implement the methodology described here may discover additional benefits and will certainly uncover additional issues which must be addressed.

	"Safe" Zone	Flag	Alarm
LOC	Under 50	50 - 100	Over 100
Effor	Under 50000	50000 - 100000	Over 100000
CC	Under 10	11-20	Over 20

Figure 1: Threshold analysis example using three code metrics (Lines of Code, Halstead's effort, and McCabe's Cyclomatic Complexity).

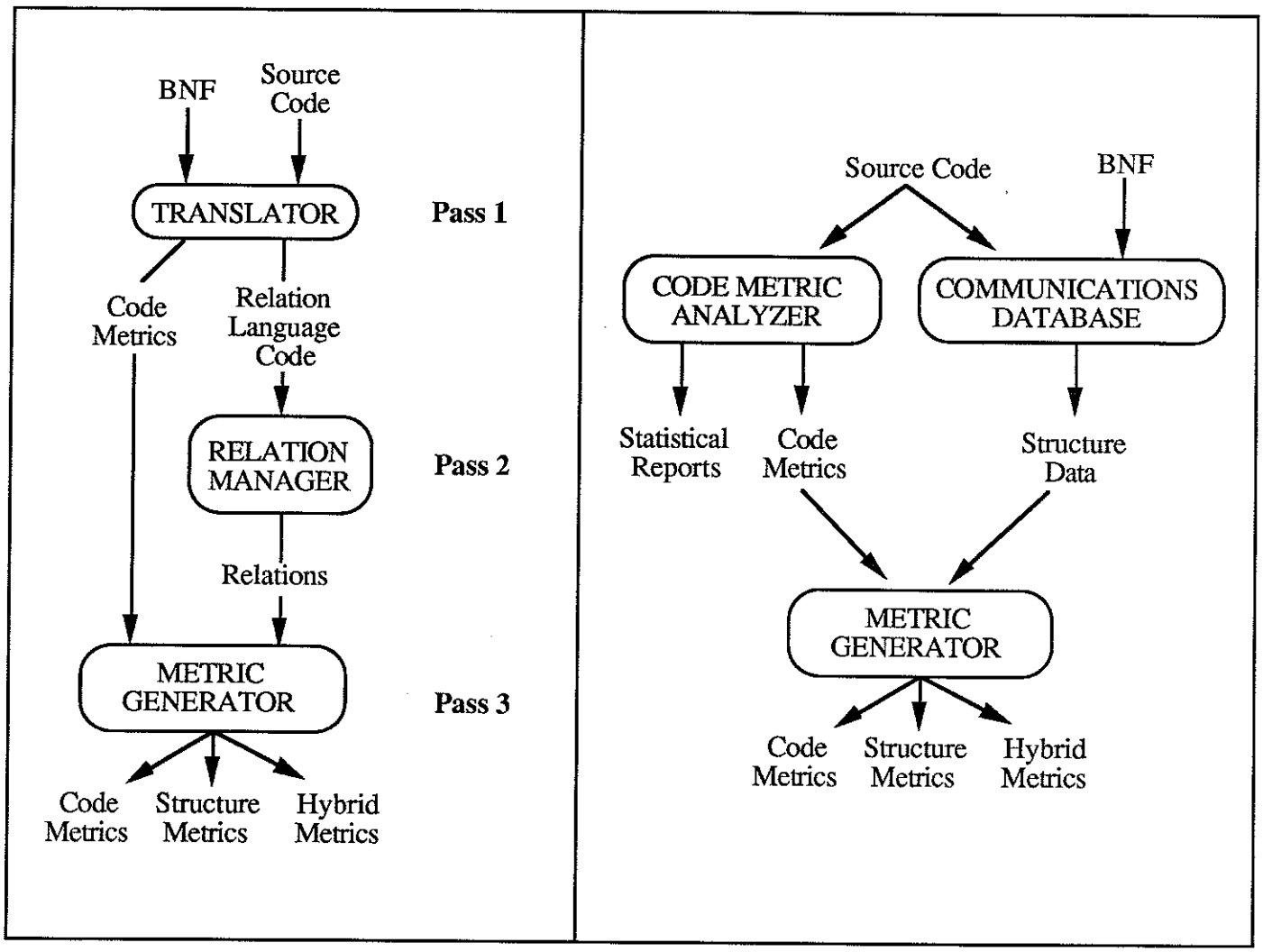
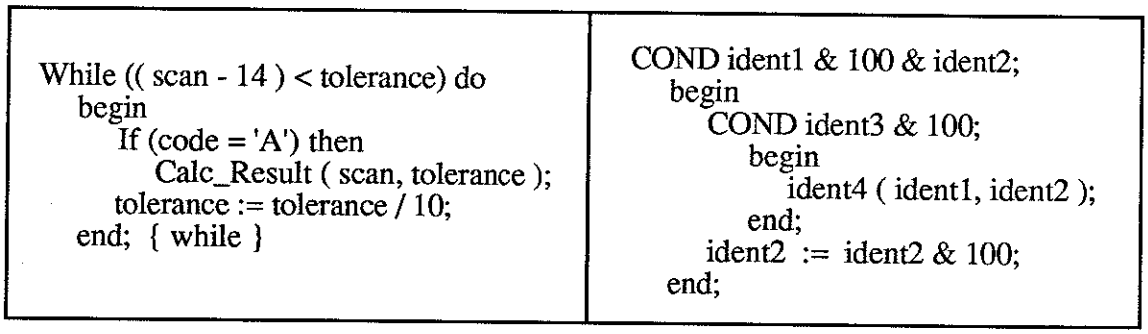


Figure 2: Metric collection tools from (a) [WAKS88] and (b) [LEWJ89].



```
While (( scan - 14 ) < tolerance) do
   begin
      If (code = 'A') then
         Calc_Result ( scan, tolerance );
      tolerance := tolerance / 10;
   end;  { while }
```

```
COND ident1 & 100 & ident2;
   begin
      COND ident3 & 100;
         begin
            ident4 ( ident1, ident2 );
         end;
      ident2  :=  ident2 & 100;
   end;
```

Figure 3: A Pascal code segment and its relation language translation.

References

[ARTJ89] Arthur, J.D., Stevens, K.T., "Assessing the Adequacy of Documentation Through Document Quality Indicators," IEEE Conference on Software Maintenance, October 1989, pp. 40-49.

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

[BROF87] Brooks, F.P., "No Silver Bullet: Essence and accidents of Software Engineering," Computer, April 1987, pp. 10-19.

[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, October 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, September 1981, pp. 510-518.

[HENS88] Henry, S.M., "A Technique for Hiding Proprietary Details While Providing Sufficient Information for Researchers," Journal of Systems and Software, Vol. 8, 1988, pp. 3-11.

[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.

[LEWJ89] Lewis, J.A., Henry, S.M., "A Methodology for Integrating Maintainability Using Software Metrics," IEEE Conference on Software Maintenance, October 1989, pp. 32-39.

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

[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, October 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.