Managerial Issues in Developing a Quality Metrics Program

By John A. Lewis and Sallie M. Henry

TR 90-50

Managerial Issues in Developing a Quality Metrics Program

by

John A. Lewis

and

Sallie M. Henry

Computer Science Department
Virginia Tech
Blacksburg, Virginia 24060

Internet: lewis@vtopus.cs.vt.edu

Managerial Issues in Developing a Quality Metrics Program

(abstract)

Software quality metrics are used to determine error prone code due to excessive complexity. These results can be used to guide testing efforts and predict future maintenance needs. However, implementing a quality metrics program involves many subtle issues which complicate the development and use of a metrics methodology. Many of these issues are managerial in nature. This paper examines some managerial elements of designing and implementing a quality metrics program. Previous studies which incorporate a metrics methodology into two different commercial environments are used to demonstrate the difficulties in implementation and approach.

1. Introduction

Many approaches have been offered to control software development and 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 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.

1.1 A Quality Metrics Methodology

A quality metrics 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.

Complex code is error-prone and difficult to maintain. In a software metrics approach, 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.

Metrics can be divided into three classes: (1) code metrics, which concentrate on the amount of internal information exists for a procedure, (2) structure metrics, which deal with a procedure as it exists within the framework of the entire system, and (3) hybrid metrics, which use aspects of both code and structure metrics.

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 Sections 2-7. 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.

1.2 Two Commercial Examples

Two studies which integrate a 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.

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.

1.3 Managerial Issues

The benefits of using a 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 a metrics program into an existing development structure,
- Collecting and using historical project data,
- Handling proprietary information,
- Interpretation of metric values, and
- Proper managerial attitude toward a metrics program.

These issues are discussed in detail in the following sections.

2. 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, metric analysis 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.

3. Integration

A metrics methodology can be integrated into an 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 difficult or distracting, the "big picture" can be ignored in favor of deadlines and 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 a variety of techniques, the metric analysis can correspond to testing efforts and therefore not require unusual preparation. The analysis process itself can be automated and therefore primarily effortless. The only substantial impact is the execution of corrective actions when error-prone code is identified. As discussed in Section 1, 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.

4. Historical Data

Many metric analysis techniques are developed using historical project data. For example, one 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) [HALM77], McCabe's cyclomatic complexity (V(G)) [MCCT76], and Henry and Kafura's hybrid information flow metric weighted by effort (INFO-E) [HENS81]. 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)).

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 above equation 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 1.2 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.

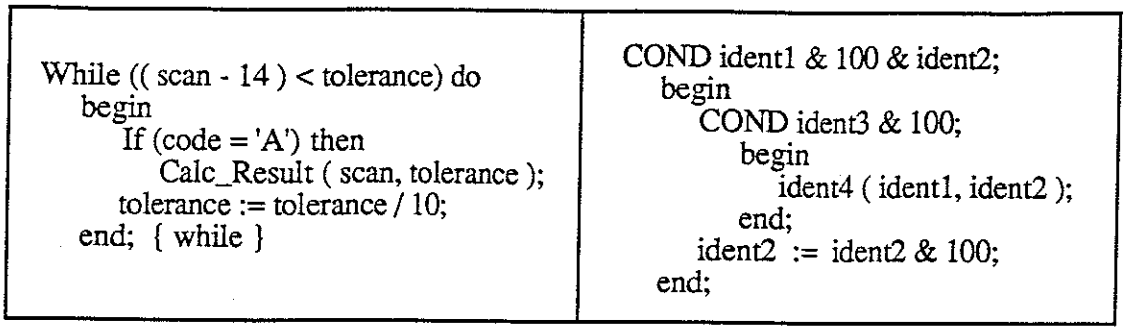
5. Proprietary Information

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 1.2 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 earlier, 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 1 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.



```
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 1: A Pascal code segment and its relation language translation.

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.

6. 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. Therefore most analysis techniques 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. Threshold analysis might raise a flag from the lines of code metric, but others might be negligible. Therefore, the technique is designed such that some consensus between the metrics must be established before concern is raised.

Furthermore, threshold analysis assumes some low value of a metric is "safe". 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 1.2 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, 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.

7. Managerial Attitude

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. 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. 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.

8. Summary

A 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.

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 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.

References

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

[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.

[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.