Avoiding metric monsters: A design metrics approach

Wayne M. Zage[†] and Dolores M. Zage

Computer Science Department, Ball State University, Muncie, IN 47306, USA

Cathy Wilburn

Northrop Grumman Corporation, Electronics Systems Division, Rolling Meadows, IL 60008, USA

Metric monsters are stumbling blocks that prevent software metrics-guided methodologies from attaining product and process improvement. Metric monsters can occur during the identification, collection or application of software metrics. In our research, we have developed and tested our design metrics over a five-year period and have found them to be excellent predictors of error-prone modules. Based on this research, we will identify some of the monsters that occur in the quantitative analyses of software and its development processes, and present our approach in formulating a design metrics model that avoids these monsters. This model consists of software tools, guidelines and actions for the application of software design metrics.

Keywords: Design metrics, design metrics analyzer, metrics model, software metrics, software quality.

1. INTRODUCTION

In software engineering, it is useful to derive analogies from other fields to gain an understanding of software development. Parallels to the industrial model for controlling processes can be drawn in an effort to improve the quality of software and the process that produces it (figure 1). The industrial model displays a process that is stable and divisible into a number of steps. During each step, the process is monitored, and measurements are taken and compared to standards for that measured characteristic [Dunn 1990]. The cornerstone of the industrial model is measurement.

This successful model can serve as the basis for a quality control model for software development. However, the software development model contains problems that can prevent it from producing the positive effects of its industrial counterpart. Whereas measurement is the cornerstone of the industrial model, measurement in the software development process can lead to problems caused by what we label *metric monsters*. As depicted in figure 2, when you are up to your neck in metric monsters,

[†] E-mail: wmz@bsu-cs.bsu.edu

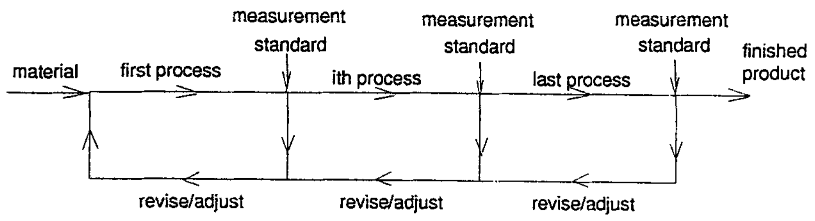

Figure 1. Industrial quality improvement model.

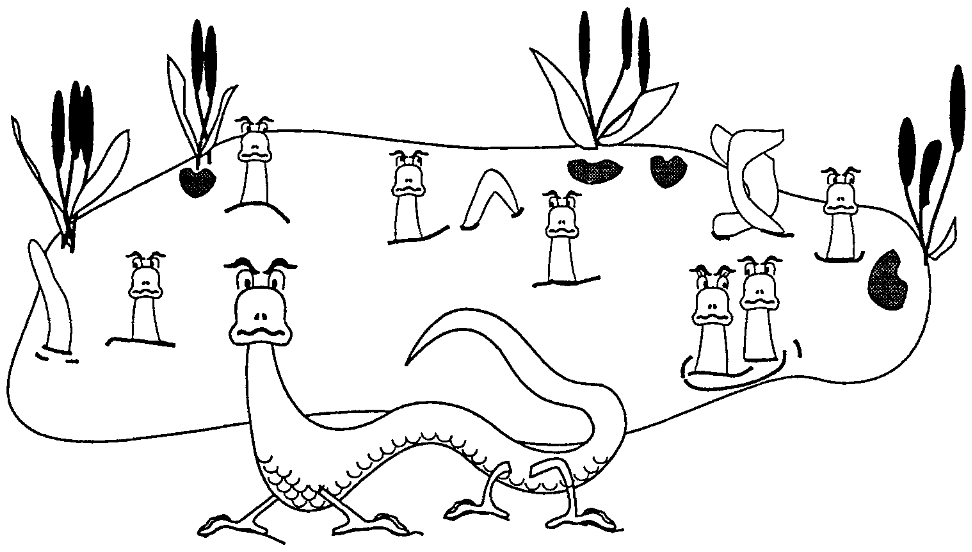

Figure 2. Metric monsters.

it is difficult to remind yourself that your initial objective was to drain the swamp, or in the software realm, to improve the process and/or product of software development. Metric monsters are stumbling blocks in the areas of *identification*, *collection* and *application* of metrics that prevent software metrics-guided methodologies from attaining product and process improvement. Sections 2, 3, and 4 of this paper begin with a discussion of predominant metric monsters in these three areas and end with a perspective from our own research efforts on how to avoid them.

2. IDENTIFICATION OF USEFUL SOFTWARE METRICS

To apply the industrial model in figure 1 to software development, the initial constraint of establishing a stable and divisible process first must be met. This condition is satisfied by many software engineers who follow a software development model such as the classic waterfall model or an iterative prototyping approach.

Having met this requirement, a measurement/standard relationship must be established for each of the defined processes. The key issues are how does one recognize and measure when the *i*th process is complete at a certain level of quality. Many sources such as intuition, office conversations and cherished beliefs are prospective benchmarks for software developers. However, standards developed from such sources are hardly reliable. For example, conventional wisdom dictates that systems composed of small modules are more reliable than those made up of large modules, that low coupling (a measure of how much modules depend on each other) and high cohesion (a measure of how related the internal parts of a module are to each other and to the functionality of the module) always lead to less error-prone modules, and that high fan-out leads to lower-quality systems. These premises seem reasonable, but some have been proven to be incorrect under certain conditions. (See [Card *et al.* 1986; Conte *et al.* 1986; Branyan *et al.* 1987] for examples.)

Selecting metrics for software measurements that have not been validated creates a metric monster. As mentioned by Grady [1992], practices of collecting and using metrics that have not been measured and proven should not be accepted as "software engineering". A rigorous analysis that validates the metrics must exist. The metrics must be field tested and instill confidence in the practitioner with their adoption. Furthermore, once metrics seem promising based on university results, they should be evaluated in industry on large-scale software [Conte *et al.* 1986].

Other problems in choosing metrics are that the wrong characteristic of the product or the process may be measured, or the intended objective cannot be measured consistently or accurately [Grady and Caswell 1987]. Moreover, one must ask if a chosen metric is really a reliable indicator of what it purports to measure. For example, software quality has many facets that are sometimes measured by combining various quality metrics to obtain a single metric value. In this approach, liberties may be taken. Ordinal and cardinal values may be combined, tradeoff qualities may not be considered and, thus, the value of the metric can be obscured and its validity questioned.

Metrics new to a measurement program must be nurtured over time. The patience and persistence of practitioners needed to establish historical data that are required to evaluate and fine-tune the performance of the metrics often run out prematurely. The benefits of a new metric may not be recognized immediately and, therefore, management must be committed to a multi-year time frame. New metrics may not be clearly and precisely defined so as to be calculated accurately and consistently. This monster also exhibits itself in established metrics. For example, the definition of *LOC* can vary between organizations, departments, individuals and tools, which can lead to possible misinterpretation of data.

2.1 Identification of Useful Design Metrics

Currently, the most commonly used product metrics are calculated from source code. These measurements can provide developers feedback, but generally too late to easily make changes without severely impacting a product's cost and schedule. Pfleeger [1991] states that it is easier to make changes to software when it is in its abstract conceptual stage than when it is already implemented. The earlier a problem can be identified, the fewer places that need to be checked to find its cause. The ideal approach would be to collect metrics during design. However, a noticeably weak area in metrics is quality indicators for design, especially preliminary designs, as noted by Schulmeyer and McManus [1992]. Currently, managers depend on inspection and/or testing to achieve quality. In any improvement process, quality must be built into the product and not added later. Improving the design process helps to ensure that quality is built into the product as opposed to engineers attempting to test it in later [Zultner 1988].

Since it is chaotic to attempt to improve all phases of software development at once, and since the design phase provides a significant return in terms of benefits, researchers are measuring many aspects of design products. Information flow metrics, some software science metrics and McCabe's cyclomatic complexity have been used to predict code quality early in the life cycle [Henry and Selig 1990]. A software complexity metric based on module interaction has been developed which helps to analyze a software system during development in order to provide a guide to system decomposition, and ultimately lead to more reliable software [Lew *et al.* 1988]. Gibson and Senn [1989] have found that system structural differences impact software maintenance performance. Li and Cheung [1987] have suggested hybrid metrics to remedy the lack of completeness of most single-factor measures of program volume, complexity and control.

Other researchers have sought a metric which would identify problematic components early in the life cycle. Studies have shown that approximately 20% of a software system is responsible for 80% of the errors [Boehm and Papaccio 1988]. It is possible that such error-prone modules exhibit some measurable attribute to identify them as design stress points. In Basili's study [1981], the measures $V(G)$, calls, *LOC*, executable statements, revisions and Halstead's effort metric E were correlated with errors. The metric with the highest correlation with errors was the number of revisions at .67. Obviously, a revision metric does not occur early enough in the life cycle to help the software developer take corrective measures during design. In the Distos/Incas experiment, information flow measures had only average predictive capabilities [Rombach 1990].

In our design metrics research, we began analyzing software systems to determine if identifiable traits of error modules could be uncovered during design. Our selection criteria were that the metrics capturing such traits must be objective and automatable. We have developed three design metrics: D_e, D_i and $D(G)$. The *external* design metric D_e is defined as

$$D_e = e_1(\mathit{inflows} * \mathit{outflows}) + e_2(\mathit{fan\text{-}in} * \mathit{fan\text{-}out}),$$

where

inflows is the number of data entities passed to the module from superordinate or subordinate modules,

outflows is the number of data entities passed from the module to superordinate or subordinate modules,

fan-in and *fan-out* are the number of superordinate and subordinate modules, respectively, directly connected to the given module, and

e_1 and e_2 are weighting factors.

The term *inflows* * *outflows* provides an indication of the amount of data flowing through the module. The term *fan-in* * *fan-out* captures the local architectural structure around a module since these factors are equivalent to the number of modules that are structurally above and below the given module. This product gives the number of invocation sequences through the module. D_e focuses on a module's *external* relationships to other modules in the software system.

The internal design metric D_i is defined as

$$D_i = i_1(CC) + i_2(DSM) + i_3(I/O),$$

where

CC, Central Calls, is the number of procedure or function invocations,

DSM, Data Structure Manipulations, is the number of references to complex data types, which are data types that use indirect addressing,

I/O, Input/Output, is the number of external device accesses, and

i_1, i_2 and i_3 are weighting factors.

D_i incorporates factors related to a module's *internal* structure. $D(G)$ is a linear combination of the external design metric D_e and the internal design metric D_i and has the form

$$D(G) = D_e + D_i.$$

The metrics D_e and D_i are designed to offer useful information during two different stages of software design. The calculation of D_e is based on information available during architectural design, whereas D_i is calculated when detailed design is completed. In architectural design, information such as hierarchical module diagrams, data flows, functional descriptions of modules and interface descriptions are available. After completing detailed design, all of the previous information plus the chosen algorithms, and in many cases either pseudocode or a program design language representation for each module, are available.

Due to costs, not all modules in the design of a system can be reviewed. Therefore, only a small percentage of the modules in the system should be identified as potentially error-prone. In order to select a reasonable set of modules to reconsider, our standard algorithm for identifying error-prone modules begins by calculating the particular metric for each module under consideration. Then those modules whose metric value is more than one standard deviation above the mean for that metric over all the modules considered are identified as stress points. Later, when error reports are available, we determine what percent of the detected errors are actually in those stress-point modules.

Our research team has obtained excellent results in finding error-prone modules in relatively small projects by simply calculating D_e values. The 12% of the modules identified as stress points by D_e contained 53% of the detected errors over all the systems in our university database [Zage and Zage 1990]. This external metric has also been evaluated on CSC's STANFINS project. In that study, D_e performed even better than on the university database by targeting 67% of the detected errors while identifying only 12% of the modules as stress points [Zage and Zage 1993].

In our quest for an internal design metric that would be a good predictor of error-prone modules, we studied D_i with several weighting schemes. The best results occurred on our test bed of projects when we set $i_1 = i_3 = 1$ and $i_2 = 2.5$, stressing the data structure manipulation usage within the modules. When highlighting 11% of the modules as stress points, we found that 94% of the detected errors occurred in 89% of those stress points, with false positives occurring 11% of the time. Thus, using $i_2 = 2.5$ gave excellent results as a predictor of error-prone modules. We also asked how these results compared to the time-honored metrics cyclomatic complexity $V(G)$ and LOC (obviously calculated later in the life cycle) as a predictor of error-prone modules. In this study, a module was identified as a stress point by $V(G)$ if its metric value was greater than or equal to 10 (the value that McCabe thought was a reasonable upper bound [McCabe 1976]), and by LOC if the size of the module was greater than one standard deviation above the mean for all modules of that particular project. The results are summarized in table 1.

Table 1

Comparing metrics' ability to identify error-prone modules.

	$V(G)$	LOC	D_i
Modules highlighted	11%	11%	11%
Highlighted modules with errors	44%	56%	89%
Detected errors found	37%	51%	94%
False positives	56%	44%	11%

Note that the highlighted modules with errors, the percent of detected errors found and the percent of false positives were not as favorable using $V(G)$ or LOC

as when using D_i. Yet the D_i metric is available earlier than specific *LOC* counts! None of the other well-known measures that we tested performed better than the D_i metric as a predictor of error-prone modules [Zage and Zage 1990]. D_i also performed well on industrial data, where the stress points highlighted contained 74% of the detected errors [Zage and Zage 1993]. The design metrics, as shown by experimental results, give a software designer a good indication of where trouble spots exist.

The metric $D(G)$, incorporating both external and internal design metric components, has given excellent results as a predictor of error-prone modules on software projects developed at Ball State University for client-partners in industry. More specifically, the 12% of the modules highlighted as stress points by $D(G)$ contained 97% of the detected errors [Zage and Zage 1990]. $D(G)$ performed successfully on the STANFINS modules as well, always highlighting high concentrations of errors as stress points [Zage and Zage 1993].

Our design metrics approach avoids the aforementioned metric monsters of poor validation, low reliability and insufficient nurturing. These design metrics are based on the theoretical design principles of coupling, cohesion and modularity. The metric D_e is related to the coupling between modules, and D_i gives a measure of a module's cohesion. The primitive design metrics involved, namely fan-in, fan-out, inflows, outflows, central calls, data structure manipulations and input/output, were chosen from a variety of primitive metrics available in architectural and detailed design. To determine which specific primitives should be chosen to precisely define our metrics, we analyzed the errors that were occurring in both university and industry-based software projects. (Other researchers have used the correlation of metrics with program changes to locate possible design and implementation problems [Henry and Kafura 1981].) Our data showed that the predominant number of errors occurred when data were passed between modules, when modules were invoked incorrectly, when the number of central calls was relatively high, and when data structures were manipulated. These metrics were selected as the primitive components of D_e and D_i as reliable indicators of error proneness. Then, through five years of nurturing these metrics by analyzing a variety of university and industrial projects and fine-tuning their components, we developed the current formulas for D_e, D_i and $D(G)$. The important issue then was whether these composite design metrics would be excellent *predictors* of error proneness. The experimental evidence shows that they are [Zage and Zage 1993].

3. COLLECTION OF SOFTWARE METRICS

Metrics collection involves the expenditure of resources such as time, labor and money. Companies will have unwisely invested their resources if the metrics obtained are not used to achieve improvement goals. Practitioners will view such metrics as an additional burden not worth the extra effort. In an already time and resource compressed development cycle, metrics collection and analysis become monsters. As reported by Pfleeger [1993], for the Software Engineering Laboratory at the US National Aeronautics and Space Administration's Goddard Space Flight Center, incorporation of data collection and analysis adds seven to eight percent to the cost of a project.

A successful quality or productivity improvement program will require a team effort from all participants. A high degree of cooperation among configuration management personnel, project managers, software engineers, accountants and secretaries is required for measuring and improving the development process. For example, team players are necessary to bridge the gap between the metricians collecting the metrics and the software developers using the metric reports. Shared values and attitudes and a corporate commitment to a quality metrics program are required to make the effort a success.

Whereas many code analyzers exist, only a limited number of commercial tools are available that support metrics collection for other phases of the software development life cycle, as noted by Moller and Paulish [1993]. When selecting a metrics support tool, practitioners need to consider the definitions of metrics employed by the tool developers. The definitions used in the tool must coincide with the definitions in the historical database if comparisons are to be made. Moreover, many basic metrics are not independent and metrics such as time and size are often used for normalizing purposes. Therefore, the tool output must be externally readable and not stored in an undocumented internal data format. Even selecting a usable code analyzer is a monstrous task. Often, companies must develop tools internally, and companies report that approximately 10% of the total software staff can be dedicated to tool support. This is a large investment that many companies are not able to make. In a survey by Ross [1990], 108 companies responded to questions concerning metrics used in quality management. Only 4% of the companies used a metrics analyzer, and yet an overwhelming number responded that they wanted practical metrics and tools. For many, the search for an appropriate metrics tool set continues.

3.1 Collection of Our Design Metrics

Several large Ada projects from industry were offered as data for this research. Ada is increasingly being used as a design representation language for several reasons [Agresti *et al.* 1992]. First, if both design and implementation are done in the same language, the translation from one stage to the next is made more easily. Also, the Ada design syntax can be checked by an Ada compiler and tools available for Ada can be used to evaluate the designs.

To make our analyses more efficient and consistent, and to assist software engineers in determining which components in a system to re-examine, a Design Metrics Analyzer for Ada (DMAA) was developed for a Unix environment. This tool helps designers avoid the metrics monster of overexpenditure of resources in that the DMAA calculates, collects and analyzes seventy-seven metrics on each Ada module. The metrics collected are our design metrics, as well as various size and complexity metrics. The design metrics are analyzed to produce stress-point reports. Other reports that are helpful for documentation purposes and re-engineering or reverse engineering efforts are also produced by the DMAA. All of the reports are in ASCII format and are described in more detail in [Zage *et al.* 1994].

Tools must be available to successfully integrate the use of metrics into the software development process. Tools ensure consistent measurements and minimize the interference with the existing work load. Before the DMAA was available, the design metrics analysis of a 14,000-line Ada program required 420 person-hours of effort. The same program requires less than one minute using the DMAA. This tool has been used on over one million lines of Ada. It is currently being used at Northrop Grumman Corporation, Electronics Systems Division, Harris Corporation, Computer Sciences Corporation and GTE Government Systems. To support the design metric analysis of large industrial projects written in C, a Design Metrics Analyzer for C (DMAC) has also been developed.

4. APPLICATION OF SOFTWARE METRICS

Although metrics have been shown successfully to aid in the development process, many software engineers are reluctant to use them. Software engineers fear that the metrics will be used to evaluate their performance. This can lead to ill-feelings about the use of metrics and even to manipulating numbers or the software product itself to achieve a measurement goal. Grady [1992] warns that "metrics are not consistently enough defined that anyone should consider using them to measure and evaluate people". When using metrics, engineers need to be assured that metrics will not be used to measure personnel performance so that they will be more eager and accurate in their metrics collection and analysis.

A metrics approach may claim to analyze software "faster than a speeding bullet", build productive software development environments "more powerful than a locomotive", and leap past the usual metric monsters "in a single bound". However, metrics do not always fulfill expectations and sometimes too much is expected from them as a cure-all. Boastful claims of unrealistic advertising can set up the user for disillusionment. In a metrics-based software development methodology, metrics data and analyses are not sufficient. A framework of guidelines for interpreting the data and possible actions based on the results must be provided to the practitioner. Furthermore, an understanding of the project being analyzed must factor into the interpretation of the metric results. Dunn [1990] states that "the most reliable predictors are those that are derived from personal experience". Therefore, a metrics model must include the practitioner's past experience and knowledge of the application as actions are taken.

4.1 The Application of Our Design Metrics

The foundation for the successful application of software metrics is building a practical metrics model. We view a metrics model as having four main components:

metrics, *tools*, *guidelines*, and *actions*. Applying this model involves coordinating these components with software development. Tools are used to automate metrics collection and support the model to ensure efficiency and accuracy in data analysis. Guidelines define the collection points for the metrics, establish stress-point cut-off values, and employ the knowledge and experience of the designer. Based on these guidelines, defined actions are recommended.

The metrics model for the application of our design metrics is displayed in figure 3. (Compare to figure 1.) The guidelines direct practitioners to collect D_e values after architectural design and to collect D_i values after detailed design. Stress points are identified in our guidelines as those modules whose metric value is more

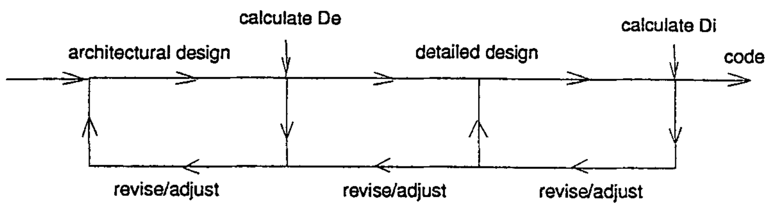

Figure 3. The design metrics quality improvement model.

than one standard deviation above the mean. Based on the identification of stress points, developers may take specific actions. For example, after calculating D_e values, software designers could modify the architectural design or progress to detailed design. Similarly, after calculating D_i values, software designers could modify the detailed design or progress to the coding phase or go back and make further modifications to the architectural design. This design metrics quality improvement model couples the information currently available during each stage of design with a measurement procedure to determine how a design is progressing.

Even in the most thoroughly tested and researched metrics model, if the actions taken as a result of the metrics data are misguided targets of finger pointing, then the model will not survive. In the design metrics model, after error-prone modules are highlighted, corrective actions are based upon the knowledge and experience of the designer. The design metrics results are used by the designers to proactively analyze their design to improve their product. By first using the metrics to identify a set of modules as potentially error-prone and then overlaying that information with personal experience and application knowledge to limit that set, the practitioner determines the appropriate actions in this model. Such actions can include looking for alternatives to a particular part of a design, assigning difficult components to experienced developers, marking a stress point for further review in the next cycle, providing extra testing effort for the indicated stress points, or simply taking no action at all. The features of our application model are summarized in table 2.

Table 2

Design metrics application model.

Metrics	Tools	Guidelines	Possible actions
D_e	DMAA	identify metric collection points	modify architectural and/or detailed design
D_i	DMAC	establish stress-point cut-off values	assign difficult components to experienced developers
$D(G)$		employ practitioner's knowledge and experience	review stress points in next design iteration
			provide extra testing effort for stress points

5. CURRENT RESEARCH DIRECTION

A software metrics model consists of metrics, tools, guidelines and actions. In our previous research, we have developed the design metrics D_e, D_i, and $D(G)$, tools to automate the metrics collection, called the Design Metrics Analyzers for Ada and C, and guidelines and actions for our model. Our current research is focused on measuring the effects of the prescribed actions on the software development process and product. The goal of this research is to evaluate the performance of our design metrics as they are applied to ongoing software development in controlled industry and university experiments.

Our research this year focuses on a design metrics analysis of a project at the Northrop Grumman Corporation. This project's design schedule will allow design changes based on our metrics data. We are using the first integration phase of approximately 100,000 lines of Ada as the control system and a second integration phase, an additional 70,000 lines of Ada, as the experimental system. Our approach permits us to establish a baseline of design and implementation defect densities and their correlations to the design metrics prior to the introduction of these metrics into the second phase.

We have the rare opportunity to apply the metrics, as they are theoretically intended to guide the design process, to industrial projects as well as to university-based projects in this coming year. Based on the outcome of this analysis, we may refine the design metrics, review our guidelines for applying these metrics to software designs, and quantify the impact of the actions taken in our model.

ACKNOWLEDGEMENTS

This research was supported, in part, by a grant from the Software Engineering Research Center (SERC) at Purdue University, and a National Science Foundation

Industry/University Cooperative Research Center grant (NSF Grant No. ECD-8913133). We thank the SERC industrial affiliates Northrop Grumman Corporation, Electronics Systems Division, Harris Corporation, Computer Sciences Corporation, Magnavox Electronic Systems Division, and the US Army Research Laboratory – Software Technology Branch, for providing our research team with large-scale software for our design metrics analyses. We also thank Pewter Graphics, Muncie, IN for drawing our metric monsters.

REFERENCES

Agresti, W.W., W.M. Evanco, M.C. Smith, and D.R. Clarson (1992), "An Approach to Software Quality Prediction from Ada Designs", Technical Report RL-TR-92-315, Rome Laboratory, Griffiss, AFB, NY.

Basili, V. (1981), "Evaluating Software Development Characteristics: Assessment of Software Measures in the Software Engineering Laboratory", In *Proceedings of the Sixth Annual Software Engineering Workshop*, SEL-81-013, National Technical Information Service, Springfield, VA.

Boehm, B. and P. Papaccio (1988), "Understanding and Controlling Software Costs", *IEEE Transactions on Software Engineering SE-14*, 10, 1462–1477.

Branyan, E., L. Thiel, and R. Rambo (1987), "Addendum to Establishment and Validation of Software Metric Factors", presented at the Fourth Annual National Joint Conference on Software Quality and Productivity, Washington, DC.

Card, D.N., V.E. Church, and W.W. Agresti (1986), "An Empirical Study of Software Design Practices", *IEEE Transactions on Software Engineering SE-12*, 22, 264–271.

Conte, S.D., H.E. Dunsmore, and V.Y. Shen (1986), *Software Engineering Metrics and Models*, Benjamin/Cummings, Menlo Park, CA.

Dunn, R.H. (1990), *Software Quality, Concepts and Plans*, Prentice-Hall, Englewood Cliffs, NJ.

Gibson, V.R. and J.A. Senn (1989), "System Structure and Software Maintenance Performance", *Communications of the ACM 32*, 3, 347–357.

Grady, R.B. and D.L. Caswell (1987), *Software Metrics: Establishing a Company-Wide Program*, Prentice–Hall, Englewood Cliffs, NJ.

Grady, R.B. (1992), *Practical Software Metrics for Project Management and Process Improvement*, Prentice–Hall, Englewood Cliffs, NJ.

Henry, S. and D. Kafura (1981), "Software Structure Metrics Based on Information Flow", *IEEE Transactions on Software Engineering SE-13*, 5, 510–518.

Henry, S. and C. Selig (1990), "Predicting Source Code Complexity at the Design Stage", *IEEE Software 7*, 2, 36–43.

Lew, K., T.S. Dillon, and K.E. Forward (1988), "Software Complexity and Its Impact on Software Reliablity", *IEEE Transactions on Software Engineering SE-14*, 11, 1645–1655.

Li, H.F. and W.K. Cheung (1987), "An Empirical Study of Software Metrics", *IEEE Transactions on Software Engineering SE-13*, 6, 697–708.

McCabe, T.J. (1976), "A Complexity Meassure", *IEEE Transactions on Software Engineering SE-2*, 4, 308–320.

Moller, K.H. and D.J. Paulish (1993), *Software Metrics, A Practitioner's Guide to Improved Product Development*, IEEE Computer Society Press, Chapman Hall, New York, NY.

Pfleeger, S.L. (1991), *Software Engineering, The Production of Quality Software*, Second Edition, MacMillan, New York, NY.

Pfleeger, S.L. (1993), "Lessons Learned in Building a Corporate Metrics Program", *IEEE Software 10*, 3, 67–74.

Rombach, H.D. (1990), "Design Measurements: Some Lessons Learned", *IEEE Software 7*, 2, 17–25.

Ross, N. (1990), "Using Metrics in Quality Management", *IEEE Software 7*, 4.

Schulmeyer, G. and J. McManus, Eds. (1992), *Total Quality Management for Software*, Van Nostrand Reinhold, New York, NY.

Zage, W.M. and D.M. Zage (1990), "Relating Design Metrics to Software Quality: Some Empirical Results", SERC-TR-74-P, Software Engineering Research Center, Purdue University, West Lafayette, IN.

Zage, W.M. and D.M. Zage (1993), "Evaluating Design Metrics on Large-Scale Software", *IEEE Software 10*, 4, 75–81.

Zage, W.M., D.M. Zage, and C. Wilburn (1994), "Achieving Software Quality Through Design Metrics Analysis", In *Proceedings of the Twelfth Annual Pacific Northwest Software Quality Conference*, Portland, OR.

Zultner, R. (1988), "The Deming Approach to Software Quality Engineering", In *Quality Progess*, American Society for Quality Control, Inc., Milwaukee, WI, pp. 58–64.