Scientific Research Publishing

Journal of Software Engineering and Applications, 2021, 14, 389-399
https://www.scirp.org/journal/jsea
ISSN Online: 1945-3124
ISSN Print: 1945-3116

Metric Based Evaluation and Improvement of Software Designs

Charles W. Butler

Department of Computer Information Systems, College of Business, Colorado State University, Fort Collins, USA
Email: charles.butler@colostate.edu

How to cite this paper: Butler, C.W. (2021) Metric Based Evaluation and Improvement of Software Designs. *Journal of Software Engineering and Applications*, **14**, 389-399.
https://doi.org/10.4236/jsea.2021.148023

Received: July 8, 2021
Accepted: August 20, 2021
Published: August 23, 2021

Abstract

The demand for quality software expands as the business environment grows internationally. Common to all software development methodologies is a design phase that focuses on the physical software model. Design criteria used to evaluate these models are important for refinement and improvement. The purpose of this research is to apply transformations of McCabe's cyclomatic complexity as a means for improving software design and assessing risks within a design. In this research, two metrics, the management (C_{MGT}) and maintenance (C_{MN}) coefficients are introduced, and they are used to address the architectural, size, and quality dimensions of a software design.

Keywords

Design Evaluation, Cyclomatic Complexity, Essential Complexity, Module Design Complexity, Maintenance Coefficient, Management Coefficient

1. Introduction

The need to develop quality software remains perhaps, one of the greatest challenges facing the business. The dependency upon software penetrates nearly every level of business operation from strategic planning to daily production and manufacturing. The software crisis, as it was labeled well over a decade ago, persists despite many contributions from structured, object-oriented, and agile development theory and practice. In retrospect, one might be frightened to project the status of software had these innovations not provided established methodologies for developing complex integrated systems. These methodologies are the primary tactics used to integrate quality into software and counter, or at least retard, the growing software maintenance liability being accumulated in businesses.

A major component of structured, object-oriented, and agile methodologies is a standard symbology for software specification such as data flow diagrams, entity-relationship diagrams, and Unified Modeling Language (UML) diagrams. Such graphical representations lend themselves to the concepts of decomposition and along with selected tools, contribute to less ambiguous, accurate software solutions. At the beginning of the development methodology evolution, it is likely that many professionals overstated the rate at which software would be implemented using these techniques. The underlying fact is that complex problems require time to find workable solutions and, using these development approaches, analysts are able to identify and evaluate more alternatives. In addition, no single methodology is embraced by everyone. Nonetheless, it is essential to realize the importance of standard symbology in both the past and future.

2. The Problem

Today, limitations exist to address the quality needs of software development, especially for complex systems [1]. The technical and management challenges of major software development projects are unparalleled. A major challenge in software development is to develop software with the right quality levels [2]. The challenge is not so much if the software development project is feasible but if the solution meets quality requirements such as functionality, performance, and maintainability. How does the software engineer evaluate and improve a software design? If one is an experienced software developer, one should constantly search for new ways to improve the solution for functionality, readability, performance, and maintainability [3]. When possible, thoughtful decisions about software should focus on resources in code—whether it's architectural soundness, testing, or system integration.

Methodologies are an outgrowth of the maturing principles of software engineering. Software development methodologies can potentially provide critical design management, which can be used for data collection and subsequent study of methodologies, so that their effectiveness can be improved. One of the reasons for the slow progress in the software discipline has been the lack of application of objective criteria to software development problems. Software development measurements, such as those associated with software metrics, and evaluation of their significance is a natural extension of software engineering. It is the goal of this paper to extend the application of metrics to software design evaluation and improvement.

3. Cyclomatic Complexity Based Metrics

Cyclomatic complexity, v, was introduced by McCabe in 1976 [4]. Generally, cyclomatic complexity is a measure of a design unit's decision structure. A design unit translates to a testable unit. Thus, an internal C^{++} procedure, a Java method, or an entire program can be a design unit. When a design unit's structure grows by adding decisions, then cyclomatic complexity grows. A design

unit's *v* has a lower bound of 1 (implying no decisions). Its upper limit is bounded by the design unit's number of logical decisions. Figure 1 illustrates a simple computation for *v*. The program design language (PDL) logic for procedure MAIN is shown as the flowgraph. Cyclomatic complexity is calculated as:

$$v = e - n + 2 \tag{1}$$

where *e* is number of flowgraph edges and n is the number of flowgraph nodes.

Cyclomatic complexity can also be calculated by adding 1 to the PDL's number of decision predicates. Thus, in the example, *v* is 4, three conditions plus 1. Moreover, *v* is used in structured testing methodology [4]. In order to qualify a program, a set of independent test paths, called the basis set, is executed. The size of the basis set (the number of tests) is equal to the design unit's *v*.

Essential complexity, *ev*, was also introduced by McCabe in 1976 [1]. Essential complexity is a measurement of a program's adherence to structured programming logical constructs. When a program's decision structure adheres to the logical constructs of hierarchy programming, the essential complexity is 1. When the program's logic violates structured programming logical constructs, *ev* increases. An increase indicates that program branches violate the single entry and exit construct. A program's *ev* can increase to the magnitude of its *v*. For example, if a program's *v* is 40 and all the logical branches violate hierarchy programming principles, then *ev* will be 40. Essential complexity's magnitude is important because it indicates the relative difficulty of modularizing a program. In other words, if a program's *v* and *ev* are 100 and 1, respectively, it is relatively straightforward to decompose the program into two programs, each with less complexity such as *v* and *ev* of 50 and 1 each. In contrast, if a program's *v* and *ev* are 100 and 100, then modularizing the program is not straightforward and could be considered to be treacherous.

In 1989, three new metrics, based on cyclomatic complexity, were introduced [5]. Module design, design, and integration complexities are extracted from a system's architecture design. Extending the hierarchy testing methodology, these metrics were shown to be a quantification of the integration testing requirement

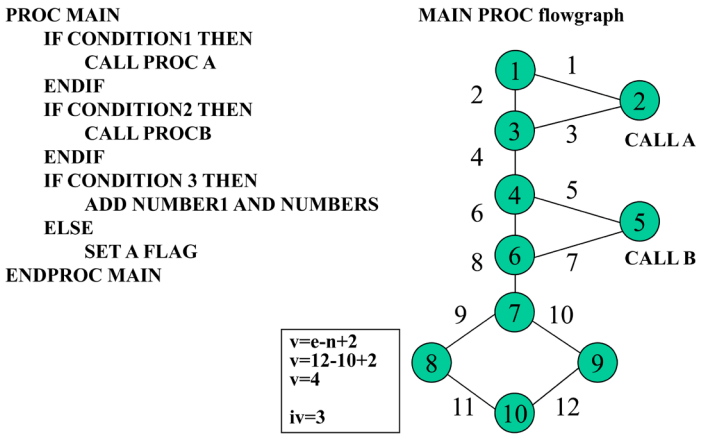

Figure 1. Cyclomatic and module design complexities.

for an entire or sub design. The first of these metrics, module design complexity, *iv*, measures a calling program's integration with its immediately called subordinates. Reconsider the PDL example in Figure 1. Module design complexity is derived using a heuristic algorithm that identifies key decisions, which controls calls to subordinate routines. In the example, cyclomatic complexity is 4, indicating that structure testing requires 4 unit test cases. The PDL's module design complexity is 3 because one decision, condition 3, does not impact the calls to subordinates. An *iv* of 3 represents the number of integration tests between MAIN and its subordinate procedures, A and B. When a program has no calling responsibility, its module design complexity is 1. In contrast, if all the program decisions are significant in calling its subroutines, then *iv* will be equal to *v*.

Design complexity, S_0, in a design volume metric. S_0 is calculated as the sum of all the *iv*'s for design units in an entire or subdesign. S_0 for the design in Figure 2 is calculated as the sum of each design units *iv*. From a testing perspective, S_0 is the total number of high-level integration tests for all calling modules. S_0 is used to determine integration complexity, S_1. S_1 is calculated as:

$$S_1 = S_0 - n + 1 \tag{2}$$

where n is the number of modules in a design.

In hierarchy testing methodology, S_1 is used to quantify the integration tests for a design. For example, Figure 2 shows how S_0 and S_1 apply to a design. In the pictured design, the S_0 and S_1 are 11 and 5, respectively. An S_1 of 5 implies that 5 test cases are needed to integrate design units A, B, C, D, E, F and G. These 5 test cases would execute various calling patterns among the design modules, under different conditions.

4. Design Improvement and Evaluation

Important evaluation attributes are objectivity and quantification. Utilized as a design concept, cyclomatic complexity inherits both attributes. At point here is not the absolute or optimum level of cyclomatic complexity. Rather, the critical point is the use of *v* as a design evaluation criterion, and the ability of a software engineer to improve a design using cyclomatic complexity-based evaluation techniques. Inherent to this approach is the published research that *v* correlates

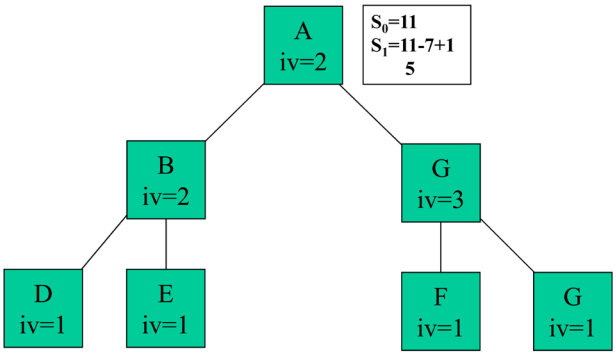

Figure 2. Design and integration complexities.

to software testability and, subsequently, quality [6] [7] [8]. If this assumption cannot be accepted, then an additional objective of the following discussion is to illustrate the effectiveness of the design evaluation techniques regardless of one's bias toward *v*.

Every implemented software system was built reflecting the formal or informal architecture defined by its creators. Design evaluation techniques work more effectively on formal design methodology. For the purposes of demonstration, cyclomatic complexity-based design evaluation is illustrated on formal design deliverables such as a hierarchy chart and PDL. All or parts of the application are transferable to other methodologies including top-down and object-oriented. The term topology will be used to refer to a design's overall breadth and depth drawing upon the analogy between the geological characteristics of landmasses and the architectural height and width of software designs.

4.1. Design Evaluation and Improvement Process

In the upcoming sections, design evaluation and improvement processes are explained and illustrated in detail. As a preface, the general process steps are outlined below.

1) McCabe cyclomatic, essential, and module design complexities are calculated.

2) Two transformations, maintenance, and management coefficients, are generated.

3) For a design architecture, cyclomatic complexity, essential complexity, module design complexity, maintenance coefficient, and management coefficient are examined identifying statistical outliers.

4) For the design architecture, cyclomatic complexity, maintenance coefficient, and management coefficient are examined for their mathematical magnitude to assess the architectural risk implication.

5) For statistical outliers within the design architecture, they are assessed to determine for what, if any, design modifications should be implemented to reduce the risk associated with the outlier's existence in the design.

The following sections illustrate the application of this general process applied to important design quality scenarios.

4.2. Size

During design, the topology of a system's design and its components is a major design challenge. Inherently, the breadth and depth of a system serve as a primary packaging limitation to the developer. For the purposes of comparison, an extreme example is illustrated in **Figure 3**. Design X_A and X_B represent two alternate designs for the same system. Theoretically, both designs perform identical functions, the same data access, user interaction, and data manipulation. Thus, they contain the same complexity from a functional perceptive. In this example, the issue is what tactics are used to manage the internal logic complexity for system X. **Table 1** illustrates a relative comparison of the two design

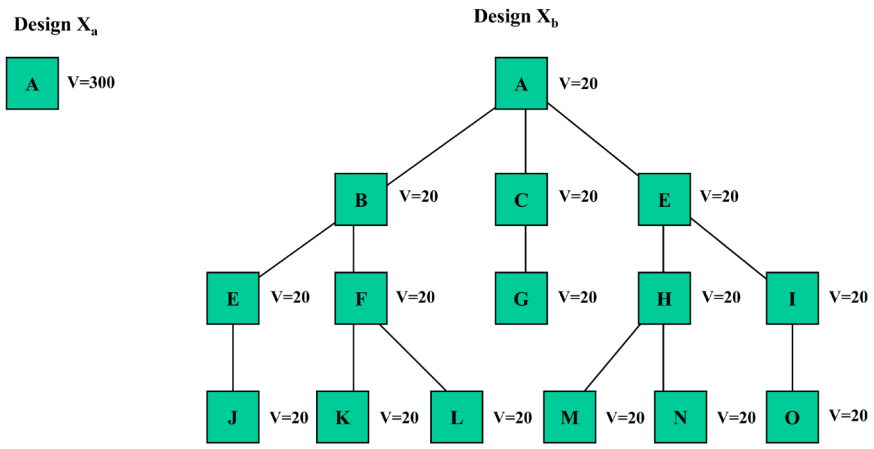

Figure 3. Alternative design topologies.

Table 1. Alternate design topologies.

Design	# Modules	Individual Module v	Total
X_A	1	300	300
X_B	15	20	300

alternatives. The architectural design for X_a is simpler, only one design unit. However, the complexity of the one design unit is exceedingly large. Thus, the design's topology reflects the tactic used to manage the system's complexity. Design X_a is a densely packaged design with all the breadth and depth contained in a single design unit. Design X_b is a design that distributes breadth and depth across 15 design units.

It is important to recall that v is a quantitative attribute that correlates to software quality. The benefits of v's application to design evaluation are not drawn from establishing an optimum value. Rather, an intuitive judgement implies that as the design unit's v increases, then understandability and maintainability will degrade. Whether a v of 10, 20 or 30 is best is not the issue. Rather design X_a's v of 300 is an indisputable high level that reduces its understandability, maintainability, and testability. The consequence is that the design X_a's module size is too large and requires further decomposition and modularity. The design challenge is determining if the design unit's size is too large and what complexity level excessive size occurs.

A more appropriate size illustration is shown in Figure 4. In this design, modules C and L v's are 40 and 44, respectively. In contrast, the remaining 13 modules v's are between 6 and 16. The size of modules C and L size are observably larger than other modules and they are candidates for critical design review and justification. Critical questions should be asked in review of modules C and L.

1) What is the impact of natural growth during system life?

2) How easily can the expected growth be managed?

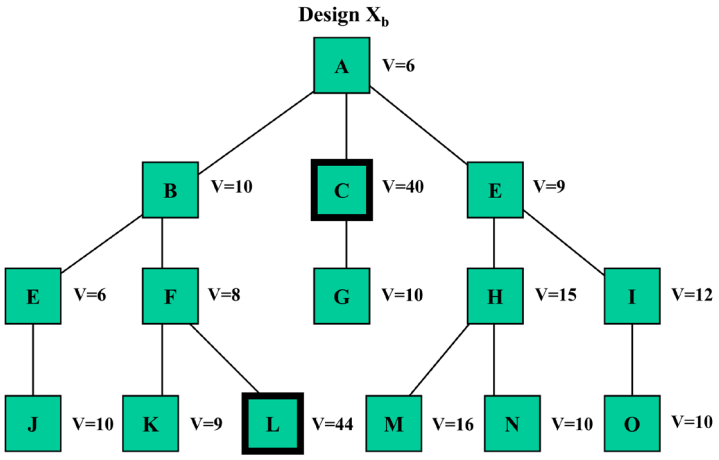

Figure 4. Design and deviating design units.

3) Is the testability of these design units, C and L, significantly reduced as growth occurs?

A true axiom for software is that it will grow over time. New functionality through maintenance and enhancement activities will add new logic, which will result in added v to a module. A high module size at the system implementation time will only increase in magnitude over the software's life and reduce future maintainability and testability.

Thus, to improve a design, design unit size, as measured by cyclomatic complexity, can be evaluated. Once quantification is achieved, improvement techniques can be applied. First, a general threshold for design units should be established. The accepted threshold should account for testability and future software growth. Second, design units whose v's are high should be subject to critical review. An objective way to identify outliers is to apply statistical quality control (SQC) techniques. The mean and standard deviation of design units can be calculated. Then, outliers that are 1, 2, and 3 standard deviations away from the mean can be identified and statistical significance can be used to determine the existence of outliers.

4.3. Maintenance Coefficient

The relative proportion of essential complexity to cyclomatic complexity serves as another evaluation criterion. The maintenance coefficient, C_{MN}, is calculated as:

$$C_{MN} = ev/v \tag{3}$$

The maintenance coefficient measures the relative importance of structuredness within a module. Therefore, an ev of 10 in a module whose v is 20 is more potentially more influential than in a module whose v is 50. Mathematically, C_{MN} is bounded by:

$$0.00 < C_{MN} \leq 1.00$$

where 0.00 is minimum maintenance impact and 1.00 is maximum maintenance impact.

Figure 5 contains maintenance coefficients for a hierarchy chart. In this example, modules D and E have cyclomatic and essential complexities of 20 and 10 and 50 and 10, respectively. Although their *ev*'s are equal, module D's maintainability is potentially reduced by the large portion of unstructuredness within the module. Using the maintenance coefficient for design evaluation, the objective is not only to achieve a low magnitude of *ev* but also a low ratio of *ev* to *v*. Thus, in circumstances where unstructuredness is justified, then its maintenance influence can be evaluated and subject to review. In addition, C_{MN} provides a time-dependent benchmark for a module's maintainability, as illustrated in Figure 6. In Figure 5, Module D originally had *v*, *ev*, and C_{MN} of 20, 10, and 0.5, respectively. In response to a maintenance change, new functions are added,

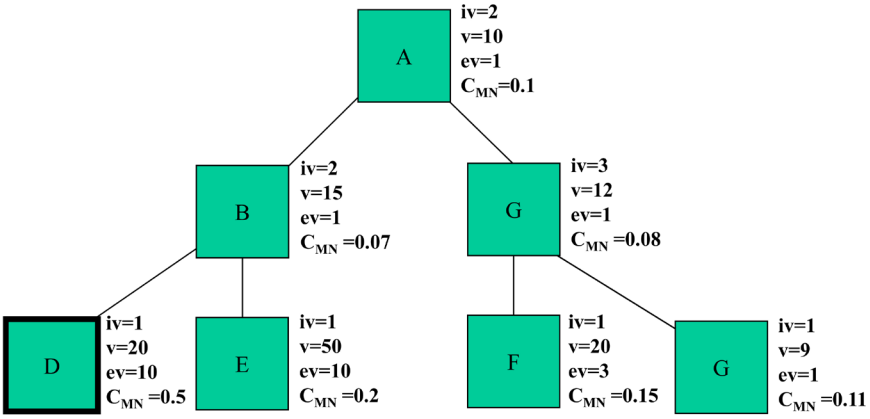

Figure 5. Comparative maintenance coefficients.

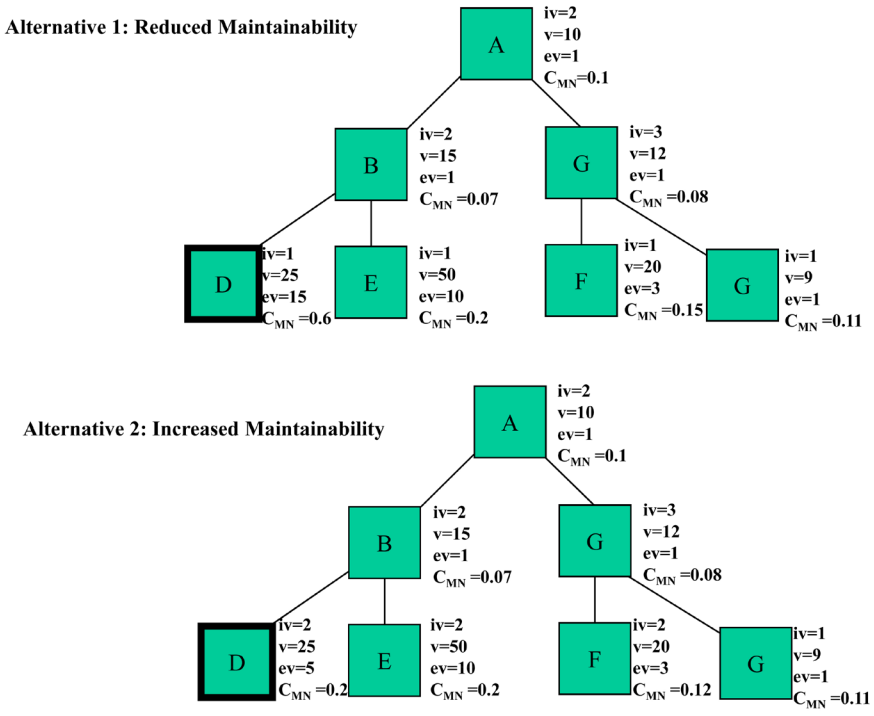

Figure 6. Change impact on the maintenance coefficient.

increasing v to 25. In alternative 1, shown in Figure 6, the changes were integrated in a non-hierarchical approach (that also impacted the existing decision structure) resulting in increased ev and C_{MN}. In alternative 2, the changes were integrated in a hierarchy manner that also positively impacted the existing decision structures. In this comparison, even though the overall cyclomatic complexity increased in both implementation approaches, alternative 2 yielded a more maintainable module.

Thus, in a manner similar to design unit size, C_{MN} can be used to improve a design. A general threshold for C_{MN} should be established. The accepted threshold should account for future software dynamics. More dynamic design units require a lower maintenance coefficient. Second, design units whose C_{MN}'s are high should be subject to critical review. As with design unit size, SQC techniques should be applied to identify outliers that are 1, 2, and 3 standard deviations away from the mean.

4.4. Management Coefficient

The ratio, module design to cyclomatic complexity, serves as an additional evaluation criterion. This ratio, called the management coefficient, C_{MGT}, is calculated as:

$$C_{MGT} = iv/v \tag{4}$$

The management coefficient, C_{MGT}, measures the relative importance of a module's decision structure in controlling execution of its subordinates. Therefore, an iv of 5 in a module whose v is 20 is more important than in a module whose v is 36. Mathematically, C_{MGT} is bounded as:

$$0.00 < C_{MGT} \leq 1.00$$

where 0.00 is no management responsibility, 1.00 is strong management responsibility.

C_{MGT} assumes greater evaluation contribution when reviewed in context to a module's location in a design. Generally, modules high in architecture should be drivers, resulting in a high management coefficient. In contrast, modules at the bottom of an architecture call nothing and have a C_{MGT} that approaches 0.

Figure 7 contains an illustration of the application of the management coefficient to design evaluations. Module A is high in the architecture with a C_{MGT} of 0.33. Since Module A is the primary driver in the design, its relative management responsibility should be higher than module I's. Therefore, C_{MGT} of A should be higher than C_{MGT} of I. This design approach implies that module A is doing additional work other than management.

Several design improvements can be made using C_{MGT}. One approach is to refactor out a module's nonmanagement work and create a new module. Thus, in Figure 7, module's A nonmanagement decision structure would be refactored out and placed in a new, subordinate design unit. The remaining logic would focus more on management and yielded a higher C_{MGT}. A second improvement technique is to layer the design and examine C_{MGT} for management levels. Generally, top-level design units should be strong managers, middle level design

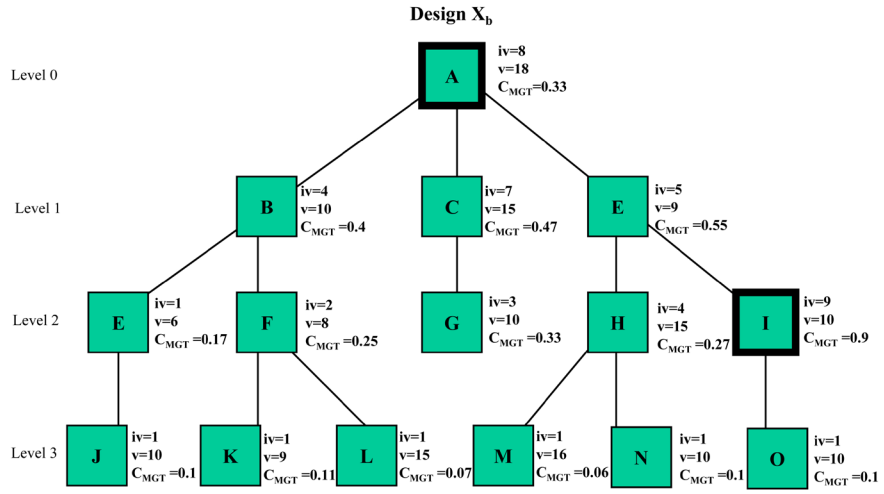

Figure 7. Management coefficients subject to review.

units should both manage and conduct work, and low-level design units should supervise and conduct large amounts of work. These design principles would be reflected in varying values across design layers. For example, proposed thresholds for C_{MGT}'s at these three management levels are as follows:

1) Top level primary management, $C_{MGT} > 0.7$;

2) Middle level management, $0.5 \le C_{MGT} \le 0.7$;

3) Low level management, $C_{MGT} < 0.5$.

A third improvement technique utilizes the SQC approach. However, when applying SQC, the analysis must be stratified within defined management layers. Thus, C_{MGT} for design units high in a design are not compared to C_{MGT} for design units low in the design.

5. Conclusions

Evaluating and improving a software design can be a subjective and risky activity. If real improvements are to be realized, then more objective criteria must be applied. The techniques defined in this paper seek to remove the subjectivity of design evaluation and improvement and replace it with objectivity. Key important tools are defined.

1) Design unit size—the cyclomatic complexity of a design unit. This measure defines the overall size of the design structure and has been statistically correlated to software quality.

2) Maintenance coefficient—the ratio of cyclomatic complexity to essential complexity. This measure defines the degree of influence of unstructuredness within a design unit. Software changes may be more costly in terms of time and manpower for design units with high maintenance coefficients.

3) Management coefficient—the ratio of module-design complexity and cyclomatic complexity. This measure defines the degree of management responsibility for a design unit relative to its position in the design architecture. Generally, design units high in design architecture should have higher management

coefficients.

The introduction of design metrics based upon cyclomatic complexity provides new approaches to design evaluation. At this point in time, there is no empirical research that correlates these design metrics to specific quality levels. However, an application of these techniques illustrates the potential design contributions outlined in this paper. Nine subsystems for a large application in a major corporate were evaluated. The driver for each subsystem was reviewed using these techniques. For the 9 drivers, the design unit size ranged from 42 to 267. The maintenance coefficient ranged from 0.51 to 0.73, and the management coefficient began at 0.08 and rose to 0.17. The drivers are exceedingly large, have high maintenance coefficients, and possess very low management coefficients. The company has found this system to be unmaintainable and has chosen to retire the existing system and replace it with a new one at the estimated cost of $6,000,000. Today, it is critical that business gains as much benefit as possible from its software. Costly maintenance and replacement must be eliminated whenever possible. If objective design evaluation and improvement contribute to this critical success factor, then the software engineering discipline has advanced another step.

Conflicts of Interest

The author declares no conflicts of interest regarding the publication of this paper.

References

[1] Stephen Jr., B., Crosson, S. and Barry, B. (2010) Evaluating the Software Design of a Complex Systems of Systems. Technical Report CMU/SEI-2009-TR-023, Software Engineering Institute, USA.

[2] Bengtsson, P.O. (1999) Design and Evaluationof Software Architecture. Research Report, University of Karlskrona/Ronneby, Ronneby, Sweden.

[3] Mitch P. (2021) 50 Tips for Improving Your Software Development Skills. https://techbeacon.com/app-dev-testing/50-tips-improving-your-software-development-game

[4] McCabe, T.J. (1976) A Complexity Measure. *IEEE Transactions on Software Engineering*, **SE-2**, 308-320. https://doi.org/10.1109/TSE.1976.233837

[5] McCabe, T.J. and Butler, C.W. (1989) Design Complexity Measurement and Testing. *Communications of the ACM*, **32**, 1415-1425. https://doi.org/10.1145/76380.76382

[6] Mannino, P., Stoddard, B. and Sudduth, T. (1990) The McCabe Software Complexity as a Design and Test Tool. *Texas Instruments Technical Journal*, **7**.

[7] Ward, T.W. (1989) Software Detection Prevention Using McCabe's Complexity Matrix. *Hewlett-Packard Journal*, **40**, 64 p.

[8] Sollers, B.H. (1992) Technical Correspondence. *Communications of the ACM*, **35**, 11 p.