A software reliability study using a complexity measure

by THOMAS J. WALSH
Control Data Corporation
Shrewsbury, New Jersey

INTRODUCTION

Software engineers face a real problem in guaranteeing that their computer programming systems under development will be able to function in a reliable manner and be easily understood, maintained and extended. A major impediment of this problem is coping with the inherent complexity of the software system in an effective way. The complexity of the computer system will defeat the designer’s efforts unless a relatively simple way is found to break the problem down in order that the resulting programs are testable and maintainable. Complex problems must be factored into smaller units to be treated by the human intelligence because man’s capacity for logically precise invention is limited. The consequence of ignoring these bounds to man’s cognitive and creative capacity was well stated by Harlan Mills of IBM:1

We often ignore the complexity of a planned program or sub-program. But when the complexity exceeds certain unknown limits, frustration ensues. Computer programs capsize under their own logical weight, or become so crippled that maintenance is impossible.

Complexity is an attribute of a computer program much like storage and speed of execution, the difference being a measurement stick of a program’s complexity has not been available. Thus, this important quantitative attribute has been generally ignored.

To appreciate the impact of ignoring a program complexity measure, let us take a brief look at the phases constituting the life-cycle of a typical software system. These phases encompass the process of design, implementation, testing and maintenance.

Scientific studies have validated the facts that at least half of the systems development time is spent in testing2 and most dollars are spent on maintaining systems.3 Figure 1 illustrates the typical break-down of software costs.4

The high cost of software is primarily due to software not being reliable enough. The key to software reliability is in the degree of precision and accuracy achieved during the design process, which has the greatest effect on the overall efficiency of the system and consequently the overall cost. This point cannot be over-emphasized. Frequently, programming systems have not been designed to facilitate testing efforts nor to constrain the impact of change. What is needed is a methodology for software system development which makes a concern for reliability an integral part of the development process, especially in the design phase. Such a methodology would consist of various tools and techniques distributed over the four previously-mentioned phases.

In recent years, what I will refer to here as “structured programming ideas” have emerged as a foundation for programming to become a science. This set of ideas has been referred to as “improved programming technologies” by IBM and as “programmer productivity techniques” by Yourdon.5

Regardless of the label, they represent several interrelated disciplines which have a natural affinity and yield noticeable benefits to the software system builder when used together in a systematic fashion.6,7

Each time a new idea is added to this evolving programming methodology, the opportunity for greater precision and reliability in programming practices increases. The analytical complexity measurement of McCabe is a new idea and deserves both recognition and membership in the promising methodology.6 The complexity measurement represents an attractive and powerful concept because of its relatively simple applicability, its direct impact on the process of design, and its mathematical origin.

Complexity can usually be calculated for computer programs by simply counting the number of decision statements and adding one. The complexity measurement focuses attention on the process of design so the function specified for the software can be intellectually gripped and performed in the simplest possible manner. This advanced design technique allows for comparing alternate designs in search of the true order that the best solution really called for. It strives to eliminate obscure structures, cumbersome decision-making processes, and overly complicated control paths. This design capability is language-independent. The mathematical origin accounts for the measure being highly correlated with the expected amount of testing work. It facilitates a more thorough and methodical testing process by yielding a minimum number of paths through a program that must be exercised in order to make testing meaningful (Structured Testing). Furthermore, it simplifies maintenance activities by the strengthening of testing and limiting the complexity of the program to be fixed. It identifies software programs that will be difficult to test and maintain and encourages the creation of a more testable and maintainable...
system. Overall, this mathematically-based measurement permits management and control of program complexity via a quantitative basis.

Figure 2 illustrates the hierarchical relationship and utilization order between top-down design, the complexity measure and structured programming. Top-down design is concerned with the "divide-and-rule" principle. It recognizes that complex problems must be factored into a combination of many small solvable problems. The complexity measure then weights the individual module's complexity to facilitate proper design, testing and maintenance. This may indicate the need for further top-down design. Finally, structured programming tackles the problem of program logic design. Figure 3 represents a survey of structured programming ideas distributed over the four phases of software development. This chart is not complete, but it is intended to be illustrative. Definitions for these techniques and related documentation techniques may be found in the Glossary.

BACKGROUND

This section's purpose is to sketch some significant historical developments in the evolution of structured programming ideas in order to properly view McCabe's complexity measure. The first major result was a paper by Bohm and Jacopini. This classic manuscript introduced organization and discipline by showing that any program, no matter how complex, can be composed in a structured manner with only three relatively simple control structures which are popularly known as "SEQUENCE," "IF THEN ELSE" and "DOWHILE" (Figure 4). An analogy of this powerful development is often made to engineering where any logic circuit can be constructed from "AND," "OR" and "NOT" gates. The importance of this result was that it mathematically dealt with the problem of complexity in control logic. The proof for the "structure theorem" is grounded firmly in mathematics. Solidly grounding viable techniques in mathematics not only increases the confidence level of present users of these techniques, but it facilitates the future development of even more powerful techniques. Although this paper was published in English in 1966, the proper recognition it deserved did not materialize until the 1970s. A major driving force in its recognition was Edsger W. Dijkstra who strongly endorsed structured programming by a famous letter in the Communications of the ACM and numerous articles.

Professor Dijkstra, historically, is a man ahead of his time whose clear thinking and proper design of programs have earned him a most influential and respected position in the computer programming profession. An underlying theme to much of his work has been the view of software as a creative branch of mathematics. Therefore, he sees the mathematical method as the most effective way for the human mind to come to grips with complexity.

Many individuals have contributed to the methodology referred to here as structured programming ideas—especially Harlan Mills, who also was an early advocate of structured programming. He provided mathematical assurance for structured programming ideas in his "Mathematical Foundations for Structured Programming." It should be noted that Mills' paper also contains the mathematical seeds that McCabe will use to simplify his theory of cyclomatic complexity. Mills has been very concerned about the problem of complexity. He views complexity as the "principal barrier to the application of computers to intelligent problem-solving." In a more recent article, he reiterates the call of Dijkstra for a mathematical basis for the practical control of computers in complex applications.

Recently, endorsements of McCabe's method as being both reasonable and intuitive have come from various sources. Glenford J. Myers of the IBM Systems Research Institute concludes his manuscript by stating:

Although it is an extremely simple concept, \( V(G) \) appears to be a practical complexity measure because it is easy to calculate, it confirms subjective opinions about complexity, and it is consistent with studies showing a high correlation between the number of decisions in a module and the modules' complexity and error proneness.

McCABE'S COMPLEXITY MEASURE

McCabe's complexity measure is a mathematical technique for calculating the logical complexity of a computer program. The complexity of a computer program is an attribute which may be assigned a number representing its logical weight. The quantitative complexity number generated is independent of the program's size, but dependent on a program's decision structure or the number of basic paths

![Diagram](https://www.computerhistory.org)
through a program. It provides a quantitative means for modularization and allows for identification of programs that will be difficult to test and maintain. Although complexity can assume a value of one to infinity, a reasonable upper limit of intellectual manageability has been placed by McCabe at ten. This number is only slightly higher than the upper bound that psychological studies confirm as the number of issues man can consider simultaneously.20

McCabe recommends that designers be required to calculate complexity as they create software programs and when complexity exceeds ten, sub-functions should be given their own procedure or the software should be redone.

In the interests of communication, this section has suppressed much of the technical mathematics which forms a solid foundation for this advanced methodology to come to grips with complexity. This would include the works of many people, including Bohm, Jacopini, and Mills.101 It would require a highly technical type of discussion outside the scope of this paper. I am very much aware of these shortcomings and can only trust this approach will not diminish the serious respect this body of knowledge so well deserves.

The following material represents some of the highlights. The theoretical basis for McCabe's complexity measure is graph theory. The following connection exists between graph theory and computer programs. Each node in the graph corresponds to a block of code in the program where the flow is sequential and the arcs correspond to branches taken in the program. Thus, all computer programs may be expressed as graphs or to be precise "program control graphs." This is an important concept because it represents a gateway through which the power of mathematical analysis may be applied to computer programs. Graph theory allows for such a graph to yield a quantitative cyclomatic complexity number via the formula:

\[ v(g) = e - n + 2 \]  

For example, in Figure 5 \( e \) (arcs) has a value of nine, \( n \)
Figure 3—Partial list of structured programming ideas.
A Software Reliability Study Using a Complexity Measure

Figure 4—Control structures of structured programming.

Figure 5—Example of complexity graph.

It should be noted that McCabe has recently proved that the structure of all programs (structured and unstructured) is equal to the number of conditions plus one. The preceding grants us the mathematical assurance to calculate the complexity of a given program either by counting the predicate nodes in the flowchart or by inspecting the source code. This allows for simplicity in the application of the complexity measure by suspending the task of drawing time-consuming graphs. In addition, the measurement process can be easily automated. In fact, McCabe has built a control structure complexity tool to run on a PDP-10 that analyzes the structure of FORTRAN programs.

AEGIS CASE STUDY

The AEGIS Naval Weapon System is an advanced shipboard combat system which is tasked with shielding the U.S. fleet. System control is governed by three high-speed general purpose AN/UYK-7 computers. An individual computer is assigned to the Radar system, the Weapons and Control System and the Command and Decision system. The heart of this large computerized system is the AN/SPY-1A three-dimensional phased array radar. The reliability study focused on eight functionally-related computer modules which constituted what is known as the software control loop and display processing of the radar. Each module was assigned with a primary function of the radar control soft-
ware. This included radar scheduling, search management, track processing, radar return processing, etc. A module itself is further divided into procedures to perform specific tasks within the module. These eight modules were composed of 276 procedures or programs which were the actual subjects of the complexity study. These procedures were written in a high-level language (CMS-2Y) and run on a 4-bay AN/UYK-7 high-speed computer. The AEGIS project utilized the software engineering techniques of top-down design and structured programming.

The methodology called for the calculation of a number of quantitative parameters for each procedure. These parameters included the number of software errors experienced, the number of source statements, the number of machine words, and a complexity number. The first parameter called for the collection and analysis of software errors detected during the development phase of these modules. The number of software errors were distilled from an existing program trouble report system. A serious effort was made to obtain only those problem reports related to software errors rather than design changes. Interviews with the programmers responsible for the code facilitated this end. The procurement of a complexity number was through McCabe's complexity measurement recommendation.

A correlation was found between a high complexity value and the occurrence of bugs for a procedure. Those procedures with a complexity greater than or equal to ten accounted for a disproportionate share of the bugs. Overall, 23 percent of the procedures accounted for 53 percent of the bugs. This fact alone is not conclusive. In general, the procedures with a complexity measurement greater than or equal to ten were also the largest users of source statements.

This meant that a correlation also existed between a high number of source statements and the occurrence of bugs for a procedure. Recognition of this phenomenon, most notably described in the *New York Times* project by IBM, has led to attempts to limit the physical size of procedures, e.g., 50 lines of source code. An obvious flaw with limiting programs solely by physical size is that it ignores the density of control structures in those 50 lines.

The principal result of the study into logical complexity occurs when the 276 procedures are divided into two groups and their respective error rates are compared. These groups are defined as those procedures with a complexity measurement less than ten and those procedures with a complexity measurement greater than or equal to ten (Figure 6). Approximately half of the actual source code is in each of these groups. Yet, those procedures with complexity greater than or equal to ten experienced over 21 percent more errors. The error rate for the group of procedures with complexity measured below ten was 4.59 errors per 100 source statements. The error rate for the group of procedures with complexity greater than or equal to ten was 5.60 errors per 100 source statements. Clearly, the effect of these error rate differences in a large software system is significant.

Now, Figure 7 illustrates what empirical studies have shown concerning the relationship between detected and undetected errors. As the number of detected errors in a piece of software increases, the probability of the existence of more undetected errors also increases. Put simply, errors come in clusters. Thus, it can be confidently predicted that when the procedures in the study enter the maintenance phase of their existence, the procedures with a complexity greater than or equal to ten will continue to experience
SUMMARY AND RECOMMENDATIONS

Reliable software is no accident. It is the residue of a collection of software engineering techniques which have a natural affinity and are distributed over the four phases of software development. The key phase is design because its effects propagate through all the other phases. A new software engineering design technique is McCabe’s quantitative complexity measure which is mathematically linked to the works of Bohm, Jacopini and Mills. The following recommendations are made as a result of my study utilizing McCabe’s complexity measure to software system builders:

1. The complexity measure should be viewed as a structured programming technique and employed with the other structured programming techniques to enhance software reliability.

2. The complexity measure should be used to create a more testable and maintainable system by warning designers when a program has become too complex.

3. The complexity measure should be used to evaluate alternate designs with the goal of finding the simplest possible solution to the problem specifications.

4. The complexity measure should be used as a more thorough and methodical testing process which quantifies the amount of work necessary for reliable testing.

5. The complexity measure should be viewed as an aid to the maintenance process via its strengthening of testing and the limiting of the complexity of the program to be fixed.

6. The complexity measure should be used on existing software to identify programs that will be difficult to maintain and extend. These programs are prime candidates for redesign.

REFERENCES


STRUCTURED PROGRAMMING IDEAS GLOSSARY

1. Chief Programmer Team—A group of computer specialists organized into a team much like a surgical team.

2. Code Walk-Throughs—A walk-through of the actual code to guarantee that it reflects the design document.
3. **Complexity Measure**—A quantitative attribute of a computer program which measures its logical weight. Mathematically expressed via formula \( v(g) = e - n + 2 \).
4. **Hipo Hierarchy Chart**—A documentation technique based on function.
5. **Design Walk-Throughs**—A walk-through of the design document checking for errors or omissions in the architecture of the design.
6. **Maintenance Walk-Throughs**—A walk-through of the proposed changes to guarantee the "fix" is not going to unintentionally impact other parts of the system.
7. **Nassi-Shneiderman Charts (Chapin Charts)**—A chart detailing the internal logic of a module.
8. **Programmer Librarian**—A designated person who handles the clerical activities of programming.
9. **Pseudocode**—An informal documentation method representing structured programming logic.
10. **Regression Techniques**—Analytical techniques which are capable of comparing different versions of the system and indicating differences.
11. **Structured Design**—A set of design guidelines to be used at the modular level.
12. **Structured Programming**—A programming technique based on the combination of three basic forms resulting in programs which can be easily read, modified and maintained.
13. **Structured Testing**—Testing guidelines which require that the number of tests not be less than the cyclomatic complexity.
14. **Top-Down Design**—A design strategy which constantly seeks to factor a problem into its smallest parts.
15. **Top-Down Implementation**—Coding in a top-down fashion or the higher level modules first.
16. **Top-Down Testing**—Testing the higher-level modules of a system before the lower-level modules.