

Appendix M

Software Complexity

Content

M.1 Introduction	M-3
M.2 Open Re-engineering	M-5
M.2.1 Common Complexity Measures	M-6
M.2.2 Complexity and Testing	M-7
M.2.3 Complexity and Re-engineering	M-8
M.2.4 Complexity and Reuse	M-8
M.2.5 Implementing A Complexity Measurement Program	M-9
M.3 Conclusions	M-9
M.4 References	M-10
M.5 Editor's Note	M-11

M.1 Introduction

Software complexity is one branch of software metrics that is focused on direct measurement of software attributes, as opposed to indirect software measures such as project milestone status and reported system failures. Current military metrics programs emphasize non-complexity metrics that track project management information about schedules, costs, and defects. While such project tracking measures are necessary to any substantial software engineering effort, they lack predictive power and are thus inadequate for risk management. Complexity measures can be used to predict critical information about reliability and maintainability of software systems from automatic analysis of the source code. Complexity measures also provide continuous feedback during a software project to help control the development process. During testing and maintenance, they provide detailed information about software modules to help pinpoint areas of potential instability. Figure M-1 shows the control flow graph of a simple, low-risk software module. Figure M-2 shows a complex, moderate-risk software module. Figure M-2 shows an extremely complex, high-risk module. Complexity metrics quantify that difference for use in software management. Measurement of software complexity provides substantial value to a software metrics program.

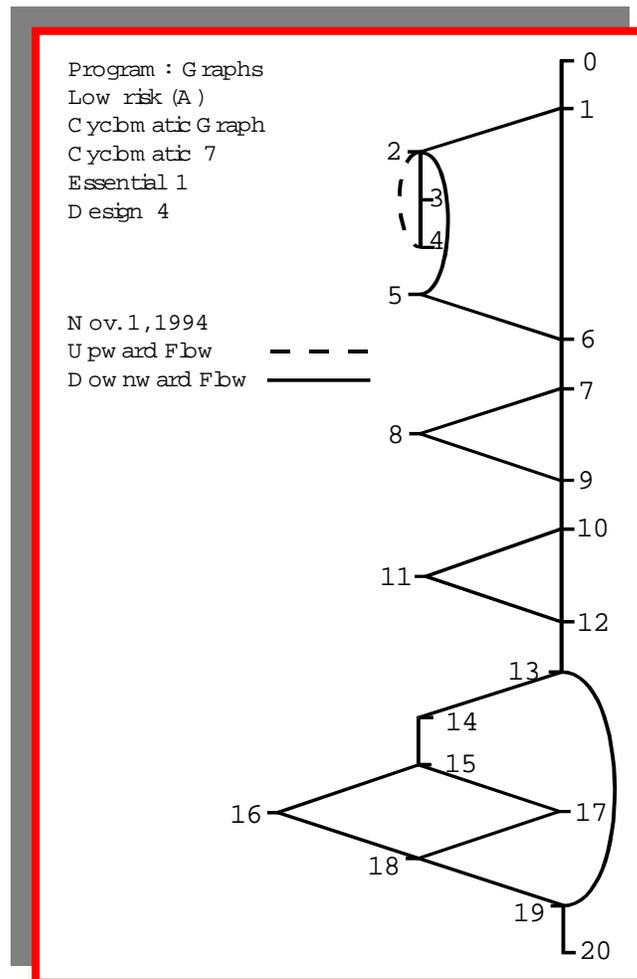


Figure M-1. Simple, Low-Risk Software Module

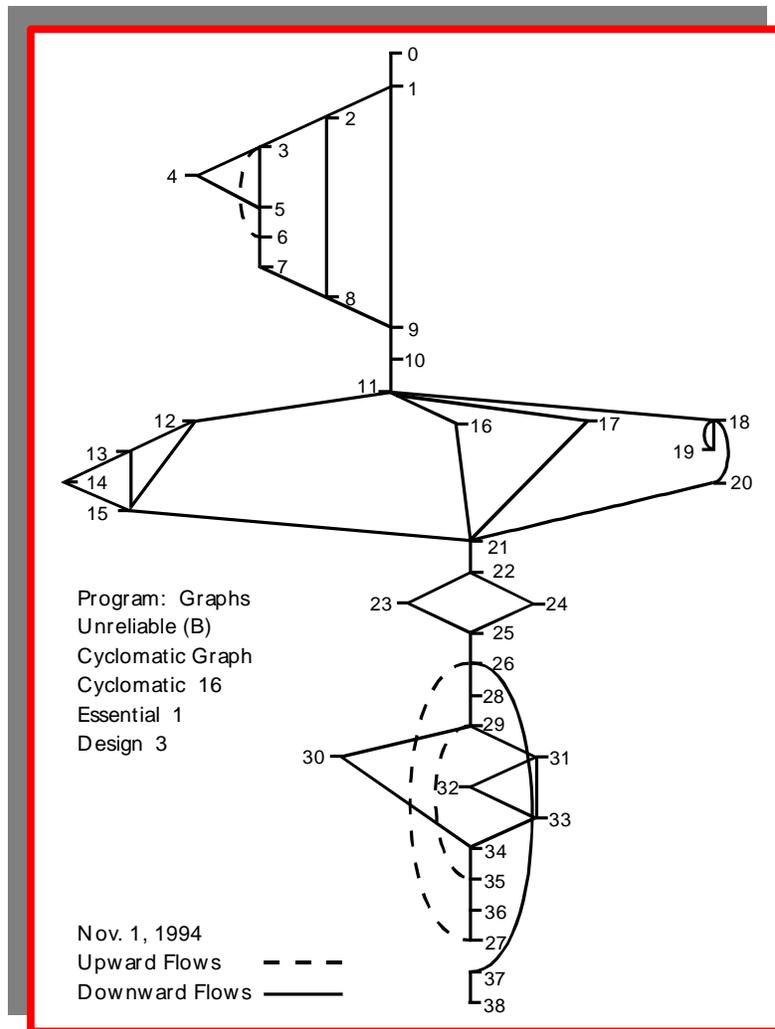


Figure M-2. Complex, Moderate-Risk Software Module

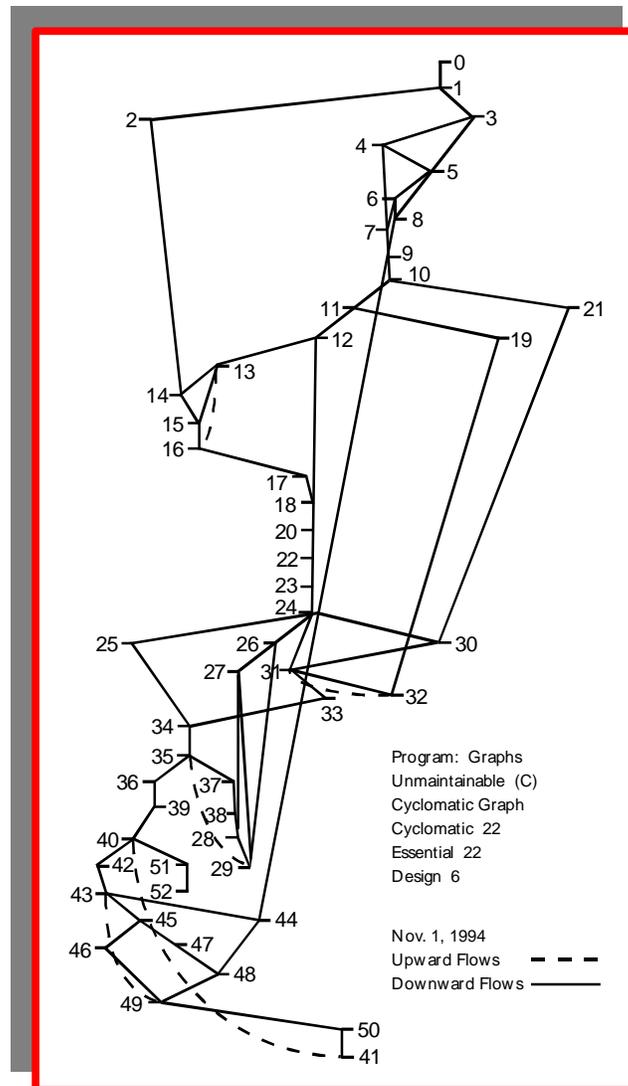


Figure M-3. Extremely High-Risk, Complex Software Module

M.2 Open Re-engineering

There are hundreds of software complexity measures, ranging from the simple, such as source lines-of-code, to the esoteric, such as number of variable definition/usage associations. It is important to select a good subset of these measures for implementation. An important criterion for metrics selection is uniformity of application. The key idea here is “*open re-engineering*.” The reason “*open systems*” are so popular for commercial software applications is that the user is guaranteed a certain level of interoperability — the applications work together in a common framework, and applications can be ported across hardware platforms with minimal impact. The open re-engineering concept is similar, in that the abstract models used to represent our software systems should be as independent as possible of implementation characteristics such as source code formatting and programming language. Complexity measurement is a fundamental application, but open re-engineering extends to other modeling techniques such as flow graphs, structure charts, and structure-based testing.

We want to be able to set complexity standards and interpret the resultant numbers uniformly across projects and languages. A particular complexity value should mean the same thing whether it was calculated from Ada source code or from Jovial. Otherwise, to get predictive benefits from the complexity measures we would have to calibrate the results based on “*similar*” projects with known outcomes, and the process becomes too subjective for effective management. The most basic complexity measure, the number of lines-of-code, does not meet the open re-engineering criterion, since it is extremely sensitive to programming language, coding style, and textual formatting of the source code. The “*cyclomatic complexity*” measure, which measures the amount of decision logic in a source code function, meets the open re-engineering criterion. It is completely independent of text formatting and is nearly independent of programming language since the same fundamental decision structures tend to be available and uniformly used in all common programming languages. The software functions represented in Figures M-1, M-2, and M-3 have cyclomatic complexity measures of 7, 16, and 22 respectively.

Certainly, there are valuable complexity measures that are not “*open*.” For example, the amount of access to global data elements is very useful in managing C projects, even though that measure is useless for COBOL in which all data is global. However, as a foundation for a complexity measurement program, it is best to concentrate on measures that can be applied consistently across projects and languages. That way, the same interpretations and methodology can be used without having to perform applicability assessments for each project.

M.2.1 Common Complexity Measures

We’ve already discussed lines-of-code, which is about the weakest complexity measure in common use. A refinement is to count the lines of executable code, data declarations, comments, and so on individually, then look at derived measures such as the percentage of comment lines. These all suffer from the weakness that most of what is being measured is source text format, which is not an intrinsic attribute of the software implementation. Most languages have “*pretty printers*” available that reformat code to a desired set of standards, and the “*indent*” program for C has about 50 switches that configure behavior. This leads us to a related set of measures, that of coding standards conformance. If code is supposed to have a comment at the beginning of every procedure, the percentage of procedures that actually have the comment can be measured. While these source format measures give useful information for project management, they are not uniformly applicable. Their extreme sensitivity to cosmetic attributes of the source code makes them unsuitable as core complexity measures.

The Halstead Software Science metrics are a significant step up in value. [HALSTEAD77] By counting the number of total and unique operators and operands in the program, measures are derived for program size, programming effort, and estimated number of defects. Halstead metrics are independent of source code format, so they measure intrinsic attributes of the software. Since different languages have different sets of operators, it isn’t immediately obvious that these measures can be applied across languages, but there’s a “*language level*” measure that can help with conversion. Halstead metrics are a bit controversial, especially in terms of the psychological theory behind them, but they have been used productively on many projects. The main drawback is that the mathematical formulas of the major Halstead metrics are significantly removed from the code, so there isn’t a strong prescriptive component. You can identify code as potentially unreliable, but the Halstead theory doesn’t say much about how to test it or how to improve it. Also, and this gets back to uniformity of application, there aren’t any established threshold values for what constitutes dangerous software; you’re pretty much on your own when deciding what values constitute unacceptable risk. Despite these drawbacks, Halstead metrics are very useful for identifying computationally-intensive code with many dense formulas, which represent potential sources of error that other complexity measures are likely to miss.

The McCabe cyclomatic complexity measure is so versatile and widely used that it is often referred to simply as “*complexity*,” and we recommend it as the foundation of any software complexity program. [McCABE76] Since it is based purely on the decision structure of the code, it is uniformly applicable across projects and languages and is completely insensitive to cosmetic changes in code. Many studies have established its correlation with errors, so it can be used to predict reliability. More significantly, studies have shown that the risk of errors jumps for functions with a cyclomatic complexity over 10, so there’s a validated threshold for reliability screening. Also, this assessment can be performed incrementally during development and can even be estimated from a detailed design. For an individual software module, the programmer can easily calculate cyclomatic complexity manually by counting the decision constructs in the code. This allows continuous control during a project, so that unreliable code is prevented at the unit development stage. Compliance can be verified at any stage of the project using automated tools. A final benefit of cyclomatic complexity, which we will discuss in more detail later on, is that it gives a precise verifiable testing prescription — the more complex and therefore error-prone a piece of software is, the more testing it requires.

There are several specialized McCabe metrics that are derived by calculating cyclomatic complexity after all control structures satisfying certain properties have been ignored. These metrics can thus be viewed as refinements of cyclomatic complexity for specific applications. The most widely used of these specialized metrics is “*essential complexity*,” which measures the amount of unstructured decision logic in software. Unstructured code, typically caused by using “*goto*” statements or breaking out of loops, is harder to understand and maintain than well-structured code. This is because control structures that interact in unstructured ways cannot be decomposed, understood, and modified in isolation. Essential complexity is a widely used measure of maintenance risk, and a threshold value of four is typical for quality screening. Also, while cyclomatic complexity increases gradually when code is added during maintenance, essential complexity can increase dramatically by the addition of a single software patch. The patched code then becomes a source of risk for future maintenance. Using essential complexity to screen modules after each modification during maintenance can manage this risk.

As such, essential complexity is a good supplement to cyclomatic complexity as a cornerstone of a complexity measurement program. Although Figures M-1 and M-2 both have high cyclomatic complexity, Figure M-3 has high essential complexity and thus carries a significantly higher maintenance risk. Two other McCabe complexity variants, design complexity, which measures the amount of interaction between decision logic and subroutine calls, and data complexity, which measures the amount of interaction between decision logic and data references, are related to integration testing and design coupling. [McCABE89] These metrics are suitable for inclusion in a mature software complexity measurement program.

M.2.2 Complexity and Testing

The Structured Testing methodology is based on cyclomatic complexity, in the sense that the cyclomatic complexity is the number of tests required. [McCABE82] Given the correlation of complexity with errors, this is a desirable result since we want testing effort to be proportional to complexity. Many other coverage-based testing techniques, from the simple ones such as statement coverage to the complicated ones such as testing all data definition-usage associations, do not have this property. You could have arbitrarily complex software with lots of statements and data associations and still satisfy those other testing criteria with one or two tests, or you might require lots of tests. With cyclomatic complexity and Structured Testing, you know in advance exactly how many tests you’ll need, so you can do detailed test planning and manage the schedules, costs, and risks associated with unit testing. Design complexity provides similar benefits for integration testing.

However, the connection between complexity and testing goes much deeper than the number of tests. From mathematical analysis, we know that the cyclomatic complexity gives the exact number of tests necessary to test each decision outcome in a function independently. The Structured Testing methodology says that we should run such a set of tests. Thus, we're not just testing statements or decisions individually; we're verifying the interactions between different parts of decision logic. In the underlying mathematical model, we can construct any path from a combination of the tests we are required to run during testing, so we're likely to detect any sources of potential error. There are techniques to calculate a set of test paths manually from the source code, and automated tools can verify that a satisfactory set of paths has actually been run during testing. The number of independent decision outcomes exercised then becomes a dynamic metric, and testing progress can be measured and managed as this number approaches the cyclomatic complexity.

M.2.3 Complexity and Re-engineering

One of the most difficult tasks in software is maintaining a system without knowing the physical design of the code and how it relates to the original abstract design. For a large system, design documentation only takes you so far, then you have to work with the code. Not only does this entail risk in terms of introducing errors due to misunderstanding code, but in the absence of complexity analysis this is unmanageable risk. The scheduling and costing problems are almost as bad, since on the surface the code and documentation give very little indication of how big a particular maintenance task really is. Complexity analysis is a critical component of successful scheduling and risk management in a re-engineering environment.

Studies confirm that cyclomatic complexity is significantly correlated with debugging time, to a much greater extent than lines-of-code. [SHEPPARD81] Cyclomatic complexity has also been used successfully as the core metric of formal re-engineering cost models, and this is an area where a lot more work remains to be done. [DeFEE94] Although cyclomatic complexity is a good foundation and has been used in numerous case studies, for something like formal estimation we should work towards including a representative mix of complexity measures such as essential complexity and the Halstead metrics. Even the number of lines-of-code has a solid place in software management — complexity metrics don't replace your current system of software controls; they just add a new dimension of predictability, reliability, and risk management to your software process.

M.2.4 Complexity and Reuse

There's a lot of redundant code in software systems. This code duplicates the functionality and in many cases the actual implementation of other code in the system. The redundant functions tend to be maintained individually, so they diverge, and there's an enormous proliferation of errors. Redundant code is a particular risk on systems that are funded by the line-of-code, as we've seen when doing Independent Verification and Validation. It's definitely to our advantage to locate and eliminate redundant code, so that we can increase the amount of reuse and reduce the total complexity of our software. Complexity analysis can provide a lot of support. One important observation is that independent implementations of the same functionality tend to have similar control flow structure. Therefore, we can use complexity measures as a screen to identify sets of software that are potentially redundant. Using the cyclomatic and essential complexity measures to identify candidate redundant modules then proceeding to examine the full flow graph diagrams and source code, a significant amount of redundant code can be removed, with resultant benefits to system size and stability. [WILLIAMSON93]

So, complexity measurement can help us find redundant code during maintenance. But what about preventing it during development? There are many products that locate reusable code in databases, usually based on matching text in a functional description with requirements characteristics. These techniques are valuable, but are limited by the amount of effort put into documenting the code in the repository. A supplementary approach based on complexity measurement can be used with an arbitrary collection of code with no documentation or database indexing overhead. The key lies in estimating the complexity metrics of the desired component from the design or pseudo-code, and then searching the source code database for code with similar metrics. For this application, a wide variety of metrics are useful.

“Open” metrics are still important to find existing code in multiple languages, but if all you’re looking for is Ada, you can get a lot of benefit out of measuring specific language constructs. The main requirement for using complexity measures to find reusable code is that the range of the complexity measure can be predicted from the design specification for the code. For example, you might know that a particular routine should have cyclomatic complexity between five and eight, have 20 to 50 lines-of-code, not contain any exception handlers, and contain exactly one loop. Then, just as with a text-oriented database search, you get the number of software functions that match your criteria, and you can refine or relax the criteria until you get a reasonably sized list of candidates. At that point, you can look at the implementations and possibly save a lot of work with pretty much no extra overhead. This is just-in-time reuse, and complexity measurement provides the technology. The only organizational overhead is running a complexity measurement tool over the source code, which will be done anyway, and wasteful development of redundant code is avoided.

M.2.5 Implementing A Complexity Measurement Program

Complexity measurement is such a large and powerful area that it’s tempting to assess hundreds of potential metrics, run pilot projects to assess potentially useful metrics, mandate data collection, correlate metrics with project performance, and eventually have a committee produce a complexity measurement policy. This doesn’t work. It takes years to start getting value out of that kind of process, and we need to use complexity analysis to help manage projects right now.

The best way to implement a complexity measurement program is to start small. Collect data on a wide variety of metrics, but pick a small, validated, intuitive set of metrics to actually apply. Continue to use lines-of-code, and add cyclomatic complexity and essential complexity. Train the developers to calculate complexity by hand, and use tools to automate the process. Start using the complexity threshold of 10 immediately to improve software reliability. Start evaluating test plans in terms of complexity to make sure that error-prone code gets the testing attention that it needs. Then, once the operational benefits of complexity analysis have been widely experienced, risk management models can be refined with measures such as the Halstead metrics and data complexity.

M.3 Conclusions

Complexity analysis has an extremely high payoff for the investment. Moving from counting lines-of-code to calculating cyclomatic complexity has immediate, measurable benefits in terms of risk management, reliability prediction, cost containment, project scheduling, and improving overall software quality. Unlike the number of lines-of-code, a good measure like cyclomatic complexity can be used to give an objective assessment of software that is directly comparable across different projects, coding styles, and even programming languages. This enables organization-wide standards and procedures that can bring true

repeatability and predictability to software. There are many valuable complexity metrics, and more are being developed every day, so it's important to start simple, not get overwhelmed, and build a solid complexity analysis program as a foundation for adding new metrics as their benefits are demonstrated.

Thomas J. McCabe
Voice: (410) 995-1075
Fax: (410) 995-1528
Internet: tom@mccabe.com

Arthur H. Watson
Voice: (410) 995-3770
Fax: (410) 720-0192
Internet: arthur@mccabe.com

McCabe & Associates, Inc.
5501 Twin Knolls Road, Suite 111
Columbia, MD 21045

M.4 References

- [HALSTEAD77] Halstead, Maurice H., *Elements of Software Science*, Elsevier North-Holland, New York, 1977
- [McCABE76] McCabe, Thomas J., "A Complexity Measure," *IEEE Transactions on Software Engineering*, SE-2 No. 4, pp. 308-320, December 1976
- [McCABE89] McCabe, Thomas J., and Charles Butler, "Design Complexity Measurement and Testing," *Communications of the ACM*, 32, pp. 1415-1425, December 1989
- [McCABE82] McCabe, Thomas J., *Structured Testing: A Software Testing Methodology Using the Cyclomatic Complexity Metric*, National Bureau of Standards, Special Publication 500-99, December 1982
- [SHEPPARD81] Sheppard, S., and E. Kruesi, *The Effects of the Symbology and Spatial Arrangement of Software Specifications in a Coding Task*, Tech Report TR-81-388200-3, General Electric Company, Arlington, Virginia., 1981
- [DeFEE94] DeFee, Joseph M., "Integrating Analysis Complexity Tool Output with Formal Re-engineering Estimation Processes," *Proceedings of the Second Annual McCabe Users Group Conference*, Baltimore, Maryland, 1994
- [WILLIAMSON93] Williamson, Eldonna S., "Determination of Redundancy using McCabe Complexity Metrics," *Proceedings of the First Annual McCabe Users Group Conference*, Baltimore, Maryland, 1993.

M.5 Editor's Note

This article, originally published in the December 1994 edition of *CrossTalk*, was reviewed by subject matter experts prior to publishing. One reviewer cautioned that any attempt to apply complexity measurements requires a thorough understanding of both the method and the software. When the decision is made to choose a method to measure software complexity, there is no single method that will meet every need and the use of hard and fast rules may actually increase complexity. Questions to the STSC, regarding software metrics, should be addressed to:

Software Technology Support Center
Ogden ALC/TISE
7278 Fourth Street
Hill AFB, UT 84056-5205
Voice: (801) 775-5555 DSN 775-5555
Fax: (801) 777-8069 DSN 777-8069
E-mail: consulting@stsc1.hill.af.mil
[http: www.stsc.hill.af.mil](http://www.stsc.hill.af.mil)