Software Metrics for Object-Oriented Systems *

J. Chris Coppick
The Mitre Corporation
7525 Colshire Drive
McLean, Virginia 22102
jcoppick@mitre.org

Thomas J. Cheatham
Middle Tennessee State University
Department of Computer Science
Murfreesboro, Tennessee 37132
cheatham@mtsu.edu

ABSTRACT

The application of software complexity metrics within the object-oriented paradigm is examined. Several factors affecting the complexity of an *object* are identified and discussed. Halstead's *Software Science* metrics and McCabe's *Cyclomatic Complexity* metric are extended to an object. A limit for the cyclomatic complexity of an object is suggested.

1 INTRODUCTION

Since the mid-1970s the pros and cons of software metrics have been debated. On the plus side a software metric provides a numerical measure of an aspect of software such as ***understandability***, ***testability***, or ***maintainability***. On the negative side, researchers cannot agree what factors contribute to the complexity of software or how to measure them. One thing for certain is that software systems have grown larger and more costly to maintain and the best software engineering principles must continually be applied to strengthen software development. Research into software metrics has intensified. See for example, Waguespack [12]. It is hoped that the ability to obtain accurate measures of software complexity will aid in reducing that complexity and subsequently lead to an increase in software reliability and maintainability. This will in turn reduce software costs. In fact several major corporations are using software metrics effectively in system design, defect prevention, and testing (see, for example, Mannino [7] and Ward [14]).

*This material is based on work supported by Western Kentucky University. Continuation of this work has been supported in part by Middle Tennesse State University and The Mitre Corporation.

Object-oriented software development (OOSD) has grown rapidly in popularity during the 80's. Its goal is to reduce software costs by promoting the use of sound software engineering principles like abstraction, encapsulation, and reuse. Little has been done to objectively evaluate the complexity of an object-oriented software system (OOSS). This paper discusses several factors affecting the complexity of OOSS. Two popular software metrics– Halstead's *Software Science* and McCabe's *Cyclomatic Complexity* are applied to OOSS. Each of these approaches to software evaluation has its own individual strengths and weaknesses. However, the purpose here is not to evaluate the merits of the method, but rather to define and discuss its application within the object-oriented paradigm.

Both metrics have been widely studied and applied within the traditional software development paradigm and each provides an example of one of the two basic types of metrics: *volume* and *coverage*.

Interpretations of both Halstead's and McCabe's metrics were implemented in both the traditional and object-oriented paradigm using *LISP Flavors* on a Texas Instruments' Explorer. The tools developed were used on some OOSS of different perceived complexities to help evaluate the tool and the metric in the object-oriented paradigm.

The next two sections provide a brief introduction to software metrics and object-oriented programming, respectively. The next section discusses the characteristics of OOSS which may affect complexity. The last two sections discuss the application of Halstead's and McCabe's work to OOSS.

2 SOFTWARE METRICS

Defined in relation to the programmer, *complexity* is the difficulty of such tasks as coding, debugging, testing, and modifying software [5]. The need to identify the characteristics of software which affect complexity has encouraged research and development of software metrics. One of the simplest examples of a *software metric* is the "lines of code." However, simply counting instructions has proven an unrealistic measure of complexity [10].

In the early 1970's Halstead [3] introduced several measures under the umbrella of *Software Science*. His *volume* metric seeks to represent complexity as a measure of the *size* of the program. Software Science metrics are based on counts of the number of operands and operators in a program:

n_1	=	the number of unique operators
n_2	=	the number of unique operands
N_1	=	the total number of operators
N_2	=	the total number of operands

For example, the LISP statement

```
(+ 1 (* A B) (+ A C))
```

yields:

n_1	=	2 ('+' and '*' being unique operators)
n_2	=	4 ('1', 'A', 'B', and 'C' being unique operands)
N_1	=	3 ('+', '*', and '+')
N_2	=	5 ('1', 'A', 'B', 'A', and 'C')

Parentheses are *not* counted as tokens. Halstead defined various measures based on these counts including:

vocabulary	:	$VOC = n_1 + n_2$
length	:	$LEN = N_1 + N_2$
volume	:	$V = (N_1 + N_2)\log_2(n_1 + n_2)$
potential volume	:	$V^* = (2 + n_2^*)\log_2(2 + n_2^*)$
algorithm level	:	$L = V^*/V$
programming effort	:	$E = V/L$

In the potential volume formula n_2^* represents the number of input and output parameters for the algorithm. The focus in this work is the volume which increases directly with the unique operators and operands and the *programming effort* which approximates the number of mental discriminations needed to create the program. Empirical studies have shown Halstead's predictors to have a high correlation to programming results [2]. A major problem with Software Science is the difficulty of discriminating between *operators* and *operands* [6]. In LISP, for instance, the same symbol may be both an operand *and* an operator and it may be impossible with static analysis to discriminate between the two occurrences.

```
(DEFUN ROOT-TYPE (A B C)
  (LET ((DISC (- (* B B) (* 4 A C)))
    (RTYPE 'COMPLEX)
  )
    (COND
      ((= DISC 0) (SETF RTYPE 'REPEATED))
      ((> DISC 0) (SETF RTYPE 'REAL))
    )
    RTYPE
  )
)
```

Figure 1: Sample LISP Function

McCabe's metric is based on the graph-theoretic notion of *cyclomatic complexity*. His goal was to provide a mathematically sound technique to identify modules that will be difficult to test and maintain [8]. He argues that testing and maintenance are heavily dependent on the decision structure of the software. A subroutine with 20 "IF" statements will be harder to test and maintain than a subroutine with one "IF". To get a handle on the number of "execution paths" through a program, McCabe used the cyclomatic complexity of the program's *control-flow graph*. A control-flow graph is a graphical representation of a program in which a group of statements executed in sequence form a node and a branch statement determines an edge. By using the control-flow graph a program's basic paths can be derived and used in combination to produce every possible execution path. For example, consider the simple LISP function, shown in Figure 1, which determines the type of roots of a non-degenerate quadratic $Ax^2 + Bx + C = 0$. The roots are either duplicated real, distinct real, or complex. The corresponding control-flow graph is shown in Figure 2.

There are only three (basic) paths. Thus the McCabe complexity is three. The McCabe metric is an example of a *coverage metric*; i.e., a metric which measures aspects of software like control flow or data flow. An easy way to calculate the McCabe complexity of a subroutine is to add one to the number of simple predicates which affect the flow of execution. Empirical studies have shown that 10 is a reasonable limit for the McCabe complexity of a subroutine (or function). Subroutines with a complexity larger than 10 should be examined for further decomposition. A strong correlation has been found between a high cyclomatic complexity and the occurrence of errors in a subroutine [13]. A weakness of McCabe's metric is that it does not take into account such things as nesting level of control structures [4] or

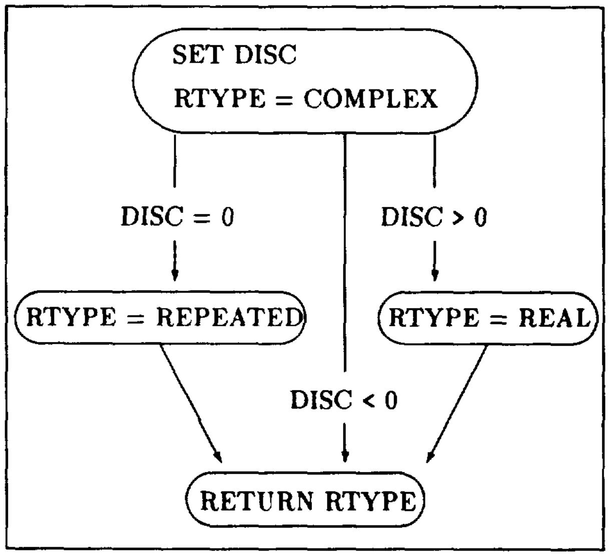

Figure 2: Sample Control-Flow Graph

program size [9].

Papers presenting variations on Halstead's and McCabe's works are abundant. Several additions and alterations exist and the two techniques have even been combined in an attempt to produce a general indicator of software complexity. Other authors have pointed out the shortcomings of either approach. This paper will not argue the pros or cons of either technique but rather it will examine how each may be applied in the object-oriented paradigm.

3 OBJECT-ORIENTED PROGRAMMING

With its emphasis on encapsulation, extendibility, and reuse, *object-oriented programming* (OOP) offers hope to developers of today's large, complex software systems. By decomposing a solution into "objects" which closely resemble the real-world entities being modeled, perhaps an advantage can be gained in understandability, maintenance, and quality. In OOP systems the basic unit of decomposition is an object which encapsulates data (called instance or state variables) and the permissible operations (called methods). In LISP Flavors an object is an instantiation of an object class called a *flavor*. A method associated with a class is defined via **(DEFMETHOD (class-name method-name) ...)**.

The concept of *inheritance* in the object-oriented paradigm is the ability to define and alter object classes by combining and adding to previously defined classes. When an object class (called the child) has inherited another class (the parent), then the child has all the same characteristics and operations as the parent. The child class may provide different versions of operations inherited from the parent, as well as providing new operations which were not defined in the parent. Some object-oriented languages only allow an object class to inherit one other class directly (*single* inheritance), while other implementations (such as LISP Flavors) provide for direct inheritance of numerous object classes (*multiple* inheritance). Inheritance is a powerful tool for abstraction and for encouraging the reuse of software by enabling the developer to combine "simpler" objects, that have already been developed and tested, to form more "complicated" objects.

To perform an operation provided by an object a *message* is sent to the object specifying the desired operation and, if necessary, any parameters. In LISP Flavors, a message

```
(SEND screen-object :draw-circle x-coord
          y-coord radius)
```

would tell an object representing the screen (*screen-object*) to execute its draw-circle method using the center (*x-coord*, *y-coord*) and the given *radius*.

Object-oriented development closely links the design and implementation phases of software engineering since objects can be designed, implemented, and tested as units without having to wait for the entire system to be designed. It is thought that the use of OOSD will help manage software complexity and enhance the usability of some metrics [15].

4 THE COMPLEXITY OF AN OBJECT

Since an *object* is the basic decomposition unit in OOSD, it is desirable to develop a measure of the complexity of an object. What is it that makes an object more (or less) complex than some other object? Is it the size (volume) of the object, the number of operations, the number or complexity of the object values, the number of object classes it inherits, or the depth of the inheritance? Certainly there is the possibility of producing a measure based on some combination of these factors.

Intuitively, the more operations available for an object, the more complex it seems. Therefore it is reasonable to want to limit the number of operations defined by any one object. At first this would seem to limit the potential usefulness of an object by saying that one object can only do so much. However, the point here is to limit the number of operations defined by an object class, not the number of operations supported by that class. If an object definition grows too complex, then that object should be decomposed into two or more objects (and potentially combined through inheritance). In this way the complexity of an object can be reduced without restricting its usefulness. Note that this is equivalent to what happens in the traditional software development paradigm; that is, when a function grows too complex it is decomposed into multiple functions. An actual numerical limit on the number of operations defined by an object will be discussed in a later section. Whatever the limit, it should be inversely proportional to the sum of the complexities of the operations. In other words, the higher the complexity of an object's operations, the smaller the number of operations which should be defined by that object.

Intuitively an object that inherits five objects each of which inherit several objects, etc., is more "complex" than a base object that inherits no other objects. However, it is not wise to include the complexity of inherited objects when measuring the complexity of another object. To do so would limit or restrict inheritance, which is one of the cornerstones of OOPS. For instance, if no object should have a complexity greater than a limit of L and an object **A** inherits an object **B** which has a complexity of K, then the complexity of **A** is actually bounded by $L - K$ not L. This is not acceptable.

Since object-oriented design is ***data centered*** rather than function centered, evaluating an object's complexity based on the data it represents and/or the number of object values utilized is a possibility. Researchers are studying measures of the complexity of both static and dynamic data components. It is considered important to obtain estimates of data complexity as early as possible in the design process [11], and this is certainly applicable to the object-oriented paradigm. However, data complexity of objects will not be addressed here.

5 APPLYING SOFTWARE SCIENCE TO OBJECTS

A tool for computing the length, vocabulary, volume, estimated program level, and estimated programming effort of objects has been implemented in LISP Flavors. Table 1 shows the results of applying the tool to the LISP source code for a set of objects of different perceived sizes and complexities. The objects themselves are part of a simple graphics editor which is provided by Texas Instruments as a demonstration of the Explorer's object-oriented and graphics capabilities. The objects represent a fair range of different characteristics, such as amount of inheritance, number of operations, etc. The tool's outputs seem reasonable in that they reflect the perceived complexities of the objects.

Since the Software-Science tool for objects operates on LISP constructs which are read as data in the form of LISP lists, pairs of parentheses are not counted as tokens. Operators follow a left parentheses as in **(+ A B)**. However not every token following a left parenthesis is an operator. For example, the token **A** in **(LET (A B C) ...)** is an operand. The LISP *FUNCTION* predicate is used to determine if a potential operator should be counted as an operator or an operand. This technique is by no means perfect, but does provide a workable compromise.

Since the volume metric is not additive [6], the volume of an object is computed relative to the entire object rather than computing the volume of the individual object components and summing. It is not difficult to prove that the volume computed relative to the entire object is always at least as large as the sum of the volume of the object's components. Halstead's ***approximation*** formula for the program level was used instead of trying to determine the exact program level. It is interesting to consider the program level of an object which has no operations. Such objects, called ***base*** objects, are not meant to be instantiated but rather to be inherited by other objects which then possibly provide operations on the base object's data. Since a base object has no function interface, its program level should remain undefined. In the table it is listed as N/A. Probably the (estimated) program effort should remain undefined for such an object, but since the tool yields results that seem intuitively reasonable, program effort for base objects is reported in Table 1.

The example data in Table 1 implies that ***programming effort***, as expected, increases along with the volume of the object. It is noteworthy that when the two objects, *CIRCLE* and *GRAPHIC-CIRCLE*, that were originally combined through inheritance were recoded as one object, named *COMBINED-CIRCLE*, the estimated program effort actually increased (see Table 2). This lends support to the intuition that increased abstraction (and inheritance) reduces programming effort. More work needs to be done in this area.

6 APPLYING CYCLOMATIC COMPLEXITY TO OBJECTS

The set of operations (methods) of an object represent a collection of algorithms which, as shown by McCabe [8], has a cyclomatic complexity equal to the sum of the complexities of each of the individual algorithms. Restricting the complexity of each method for an object to the recommended limit of 10 without placing a limit on the number of operations would not effectively limit the complexity of the object. In a sense the object's

Table 1: Software Science Measurements of Various Objects

Object	No. of Methods	Length	Vocabulary	Volume	Est. Level	Est. Effort
BASIC-GRAPHICS-OBJECT	0	12	11	41.51	N/A	22.83
CIRCLE	3	54	33	272.39	.1837	1483.05
RECTANGLE	3	57	33	287.53	.1714	1677.26
GRAPHIC-CIRCLE	2	67	40	356.56	.1111	3209.12
GRAPHICS-RECTANGLE	2	77	43	417.82	.1031	4051.61
SIMPLE-COMMAND-PANE	0	14	14	53.30	N/A	53.30
SIMPLE-GRAPHICS-PANE	0	24	21	105.42	N/A	60.61
SIMPLE-EPP-GRAPHICS-ED.	24	675	193	5124.91	.0184	277789.63

Table 2: Effect of Abstraction on the Estimated Effort of Objects

Old Objects	No. of Methods	Length	Vocabulary	Est. Effort
CIRCLE	3	54	33	1483.05
GRAPHIC-CIRCLE	2	67	40	3209.12
Total Est. Effort				4692.17
New Object				
COMBINED-CIRCLE	4	88	50	4972.96

interface is like a decision statement based on the message sent to the object. Applying a McCabe complexity limit of 10 to the interface yields a limit of 100 for the cyclomatic complexity of an object (10 methods times a complexity of 10). Rather than limit the number of operations for an object to 10, it seems more effective to limit the sum of the complexity of the object's operations to 100. Then an object can have more than 10 operations if the operations are less complex and must have less than 10 operations if some are more complex. More empirical evidence should be collected to determine if the recommended limit is optimal. Certainly, both the number and individual complexity of the object's methods must be considered. The suggested limit of 100 for the cyclomatic complexity of an object is intuitively reasonable and data collected at Hewlett-Packard [1] relating the McCabe complexity of objects with the number of known defects in those objects lends favorable support to it.

7 CONCLUSIONS

During OOSD the object becomes the basic unit of decomposition, so it is imperative that accurate complexity measures be applied to objects, and that reasonable limits for the complexity of an object be found. The use of OOP does not in itself provide for the management of complexity. It does however expedite the application of a metric in the development process since most software metrics are applied to code and an object is often designed, coded, and tested without waiting for the entire system to be designed.

Software metrics can and should be applied within the object-oriented paradigm. Factors affecting the complexity of an object have been discussed. Halstead's Software Science and McCabe's cyclomatic complexity have been applied to objects producing intuitively reasonable results.

REFERENCES

[1] Fiedler, Steven P., "Object-Oriented Unit Testing", *Hewlett-Packard Journal*, Vol. 40, No. 2, 1989, pg. 69-74.

[2] Fitsimmons, Ann and Tom Love, "A Review and Evaluation of Software Science", *Computing Surveys*, Vol. 10, No. 1, 1978, pg. 3-18.

[3] Halstead, Maurice, *Elements of Software Science*, Elsevier North-Holland, New York, 1977.

[4] Harrison, Warren A. and Kenneth I. Magel, "A Complexity Measure Based on Nesting Level", *ACM SIGPLAN Notices*, Vol. 16, No. 3, 1981, pg. 63-74.

[5] Kearney, Joseph K. and Robert L. Sedlmeyer, William B. Thompson, Michael A. Gray, and Michael A. Adler, "Software Complexity Measurement", *Communications of the ACM*, Vol. 29, No. 11, 1986, pg. 1044-1050.

[6] Levitin, Anany, "How to Measure Software Size and How Not to", *Proc. IEEE COMPSAC '86*, 1986, pg. 314-318.

[7] Mannino, Phoebe and Bob Stoddard and Tammy Sudduth, "The McCabe Software Complexity

Analysis as a Design and Test Tool", *Texas Instruments Technical Journal*, Vol. 7, No. 2, 1990, pg. 41-54.

[8] McCabe, Thomas J., "A Complexity Measure", *IEEE Trans. Soft. Eng.*, Vol. SE-2, No. 4, 1976, pg. 308-320.

[9] Ramamurthy, Bina and Austin Melton, "A Synthesis of Software Science Metrics and the Cyclomatic Number", *Proceedings IEEE COMPSAC '86*, 1986, pg. 308-313.

[10] Siyan, Karanjit S., "Coping with Program Complexity", *Dr. Dobbs Journal*, Vol. 14, No. 3, 1989, pg. 60-69.

[11] Tsai, W. T., and M. A. Lopez, V. Rodriguez, and D. Volovik, "An Approach to Measuring Data Structure Complexity", *Proceedings IEEE COMPSAC '86*, 1986, pg. 240-246.

[12] Waguespack, Leslie J. and Sunil Badlani, "Software Complexity Assessment: An Introduction and Annotated Bibliography", *ACM SIGSOFT: Software Eng. Notes*, Vol. 12, No. 4, 1987, pg. 1-40.

[13] Walsh, Thomas J., "A Software Reliability Using a Complexity Measure", *Proceedings AFIPS National Computer Conference*, Vol. 48, 1979, pg. 766-780

[14] Ward, William T., "Software Defect Prevention Using McCabe's Complexity Metric", *Hewlett-Packard Journal*, Vol. 40, No. 2, 1989, pg. 64-69.

[15] Wirfs-Brock, Rebecca and Brian Wilkerson, "Object-Oriented Design: A Responsibility Driven Approach", *OOPSLA '89 Proceedings*, 1989, pg. 71-75.