5-1990

Software Metrics for Object-Oriented Software

John Coppick

SOFTWARE METRICS FOR OBJECT-ORIENTED SOFTWARE

A Thesis

Presented to

the Faculty of the Department of

Computer Science

Western Kentucky University

Bowling Green, Kentucky

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

by

John C. Coppick

May, 1990

AUTHORIZATION FOR USE OF THESIS

Permission is hereby

☒ granted to the Western Kentucky University Library to make, or allow to be made photocopies, microfilm or other copies of this thesis for appropriate research for scholarly purposes.

☐ reserved to the author for the making of any copies of this thesis except for brief sections for research or scholarly purposes.

Signed: [signature]

Date: 5/24/90

Please place an "X" in the appropriate box.

This form will be filed with the original of the thesis and will control future use of the thesis.

SOFTWARE METRICS FOR OBJECT-ORIENTED SOFTWARE

Date Recommended 5/8/1990

Thomas J. Cheatham
Director of Thesis

John W. Cunliffe

Kenneth L. Modesitt

Date Approved May 24, 1990

Elmer Gray
Dean of the Graduate College

ACKNOWLEDGEMENTS

I wish to thank my advisor and friend, Dr. Thomas Cheatham, my friend Thomas Vaught, and my parents. Without all of their support this thesis could never have been completed. In retrospect, the next time I feel a hunger for the sweet taste of accomplishment and the support of my friends and loved ones, I think I will just take them all out to dinner.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS iii

ABSTRACT. v

Chapter

1. INTRODUCTION 1

2. SOFTWARE COMPLEXITY 4

3. OBJECT-ORIENTED PROGRAMMING 10

4. EVALUATING THE COMPLEXITY OF OBJECT CLASSES 13

5. APPLYING SOFTWARE SCIENCE TO OBJECTS 17

6. APPLYING CYCLOMATIC COMPLEXITY TO OBJECTS 24

7. SUMMARY 29

WORKS CITED 31

SOFTWARE METRICS FOR OBJECT-ORIENTED SOFTWARE

John C. Coppick

May 15, 1990

32 Pages

Directed by: Dr. Thomas Cheatham, Dr. John Crenshaw, and Dr. Ken Modesitt

Department of Computer Science

Western Kentucky University

Within this thesis the application of software complexity metrics in the object-oriented paradigm is examined. Several factors which may affect the complexity of software objects are identified and discussed. The specific applications of Maurice Halstead's Software Science and Thomas McCabe's cyclomatic-complexity metric are discussed in detail.

The goals here are to identify methods for applying existing software metrics to objects and to provide a basis of analysis for future studies of the measurement and control of software complexity in the object-oriented paradigm of software development.

Halstead's length, vocabulary, volume, program level, and effort metrics are defined for objects. A limit for the McCabe cyclomatic complexity of an object is suggested. Also, tools for calculating these metrics have been developed in LISP on a Texas Instruments' Explorer.

1. INTRODUCTION

During the past decade we have seen a marked rise in the level of interest in developing metrics to measure software complexity. As software systems have grown larger and more costly to maintain, greater emphasis has been placed on the use of software-engineering practices to strengthen software development. Consequently, research into developing software metrics and implementing these metrics into the software development life cycle has intensified. 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.

Likewise, the use of the object-oriented programming paradigm has increased dramatically in recent years. Widely viewed as a means of promoting good software-development practices, such as design modularity, encapsulation, and software reuse, object-oriented design and implementation is also considered as a means of reducing software costs. Unlike the traditional function-based paradigm, where problems are decomposed into *functional units* which operate on some defined set of data, object-oriented problem decomposition centers around developing

software representations of the *data* which then perform requested operations on themselves.

While the use of software metrics has been widely researched relative to functional paradigm of program development, there seems to have been considerably less work done in the area of applying these metrics to object-oriented software development. In this paper we will discuss some of the factors affecting the complexity of object-oriented software development and the use of a few common metrics to measure the complexity of objects. The metrics chosen for study were Halstead's Software Science and McCabe's cyclomatic complexity. Each of these approaches to software evaluation has its individual strengths and weaknesses. However, the purpose here is not to evaluate the value of a metric, but rather to analyze its application within the object-oriented paradigm. Since both the works of Halstead and McCabe have been widely researched and applied within the functional paradigm, their metrics seemed to be logical choices for an initial investigation. Also, each provides an example of one of the two basic divisions of metrics: *volume* metrics and *coverage* metrics.

Interpretations of both Halstead's and McCabe's metrics were implemented in both the functional and object-oriented paradigms in LISP on a Texas Instruments' Explorer by the author. In the Explorer environment, the object-oriented paradigm is implemented through the use of LISP Flavors. First a flavor representing an object class is defined, then methods (operations) on that flavor are defined. LISP was chosen over other available object-oriented languages (such as C++ and Smalltalk)

partially for the ability to quickly parse LISP source code using LISP. By avoiding the need for complex lexical analysis, more time was available to concentrate on the metrics themselves. The tools developed to analyze flavors include a system for examining LISP source code to collect information about the set of flavors (object classes) defined in the source code. The tools developed were used on some software objects of different personally-perceived complexities as an aid in determining if our definitions of the metrics relative to the object-oriented paradigm would produce reasonable results.

The next two chapters provide a brief introduction of software metrics and object-oriented programming, respectively. Chapter four discusses the characteristics of object-oriented programming which may affect complexity. Chapters five and six then discuss the initial applications of Halstead's and McCabe's work to the object-oriented paradigm.

2. SOFTWARE COMPLEXITY

The issue of software complexity deals with aspects of software development such as programming effort, understandability, and maintainability. Defined in relation to the programmer, complexity is the difficulty of such tasks as coding, debugging, testing, and modifying software [1]. The need to identify the characteristics of software which affect complexity has led to the research and development of *software metrics*.

One of the simplest examples of a software metric is counting the lines (instructions) of code. However, simply counting the lines of code has proven an unrealistic measure of complexity [2]. Several other metrics have been proposed for measuring different characteristics of software, but some of the most significant work to date has been performed by Maurice Halstead and Thomas McCabe.

Halstead's work, known as *Software Science*, was done in the early 1970's. His measures provide an example of *volume metrics*. Volume metrics seek to represent software complexity based on some measure of the

size of a program. Halstead based his measures on counts of the number of operands and operators in a program. The specific counts used include:

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))
```

would be interpreted as follows:*

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

Halstead defined the *volume* (V) of a program to be:

$$V = (N_1 + N_2)\log_2(n_1 + n_2)$$

The volume of a program is considered a better measure of the program size than just the number of operands and operators, since the volume increases directly with the number of unique operators and operands [3]. Halstead also expressed what is called the *potential volume* (V^*) as:

$$V^* = (2 + n_2^*)\log_2(2 + n_2^*)$$

---

*Note that in our implementation, LISP parentheses are not considered as tokens.

The potential volume represents an algorithm's shortest possible form; i.e., just the volume necessary to invoke the algorithm, as if it were built into the language. The n_2^* represents the algorithm's number of input and output parameters. The algorithmic level is then defined as $L = V^*/V$. Thus, an algorithm which can be implemented in the shortest possible form has a level of one, and as the volume necessary to implement the algorithm increases, the level decreases. The programming effort (approximating the number of elementary mental discriminations needed to create the program) is $E = V/L$ [4]. However, computing V^* is not always a straight-forward procedure. Calling syntax varies from language to language and even determining the actual number of input and output parameters can be difficult. Therefore, Halstead developed an approximation of L, called $\hat{L}$, where $\hat{L} = (2/n_1)(n_2/N_2)$. So, E could be written as $Vn_1N_2/2n_2$. There are other measures developed by Halstead which predict such things as programming time, the number of errors, and program length which will not be discussed. These predictors however have been shown to have high correlations with actual programming results [5]. One of the major drawbacks of Software Science is the difficulty which often arises when discriminating between operators and operands [6]. In LISP for instance (as will be seen in Chapter 5), it is sometimes questionable whether there is any difference between the two at all.

McCabe's work, starting in the late 1970's, is directed towards the goal of producing software modules that are both testable and maintainable. McCabe has attempted to fulfill the need for a mathematical technique that will provide a quantitative basis for measuring

modularization and identifying modules that will be difficult to test and maintain [7]. To do this, McCabe sought to measure the number of execution paths through a program. Due to the possibly unwieldy magnitude of such a number, he defined his complexity measure in terms of the basic paths of execution by using the *cyclomatic complexity* of a program's *control-flow graph*. A control-flow graph is essentially a flow chart in which sequences of statements without control-flow constructs are mapped to a single node in the graph. By using the graph a program's basic paths can be easily derived. These basic paths may be taken in combination to produce every possible path. For example, consider a function (Figure 1) which determines if the roots of a non-degenerate quadratic equation ($ax^2 + bx + c = 0$) are repeated (real), real (distinct), or complex. The

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

Figure 1: LISP Function Definition for Determining the Root Type of a Non-Degenerate, Quadratic Equation

control-flow graph for this function is given in Figure 2. Note that there are three basic paths through the function:

1 → 2 → 3 → 6
1 → 2 → 4 → 6
1 → 2 → 4 → 5 → 6

Thus, the McCabe complexity is three. McCabe's metric is considered a *coverage metric*; i.e., a metric which measures aspects of software like control flow or data flow. Skipping the mathematical foundation of McCabe's work, it may be simply stated that the *McCabe complexity* of a subroutine (function) is the number of simple predicates which affect the flow of execution plus one [8]. A McCabe complexity of 10 is generally recognized as the limit for one subroutine. Subroutines with complexities higher than 10 should be examined for further decomposition possibilities. Strong correlations have been found between a high cyclomatic complexity and the occurrence of errors in a subroutine [9]. Some of the weaknesses of McCabe's metric are that it

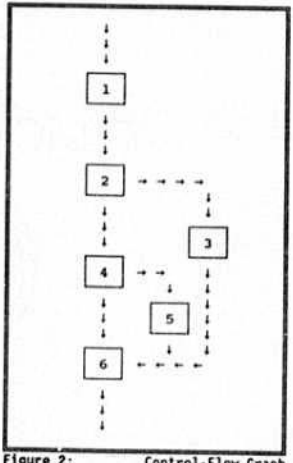

Figure 2: Control-Flow Graph for the Function REAL-TYPE

does not take into account such things as the nesting level of control structures [10] or program size [11]. Also, the metric's usefulness as an index of testing effort required (its original intended purpose) has been questioned [12].

Papers presenting variations on Halstead's and McCabe's works are abundant. Several suggest additions to the McCabe complexity or combination of Halstead's and McCabe's techniques in order to produce a general indicator of software complexity. My intent in this paper is not to absolutely prove or disprove either of these techniques' applicability to complexity measurement in general, but rather to examine how each of these techniques may be applied within the object-oriented paradigm with the goal of producing results equivalent to those obtained in the functional paradigm.

3. OBJECT-ORIENTED PROGRAMMING

Object-oriented software development provides a different methodology than the traditional functional decomposition. In the functional paradigm a problem is solved by decomposing the problem into functions which then accept a set of data and return a transformation of that data. Thus, the usual unit of decomposition or building block used to construct programs is the *function* (or subroutine or procedure). However, in the object-oriented model the basic unit of decomposition is the *object*, which is a representation of the data itself. Thus, the object-oriented technique is sometimes referred to as "data oriented decomposition" as opposed to "function-oriented decomposition." More specifically, an object represents an abstract data type which includes a data type and a related set of operations which can be performed on that type.

Objects are usually implemented by defining an object class which specifies the set of object values (also known as instance variables) and the object behavior (operations). In LISP Flavors, a class consists of a *flavor* definition and a set of associated *method* definitions which represent the operations. An object is created by instantiating a variable of a particular object-class type.

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 by 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 the direct inheritance of numerous object classes (*multiple inheritance*). Through inheritance, complicated objects can be designed by combining simpler objects which have already been designed and tested. Thus, inheritance provides a powerful tool for encouraging the reuse of software which has already been developed and tested.

An instantiated object is used by sending the object a *message* which specifies some operation for the object to perform and may contain some set of necessary parameters. In LISP Flavors, messages are sent to objects (flavor instances) by using the **SEND** function. For example,

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

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

Object-oriented development closely links the design and implementation phases of software engineering since objects can be designed, implemented and tested as separate units without waiting for an

entire system to be designed. Also, object-oriented development provides as its basis encapsulation, software reuse, and extendibility. It is thought that due to these characteristics, use of the object-oriented paradigm will help manage software complexity and actually enhance the usability of some software metrics [13].

4. EVALUATING THE COMPLEXITY OF OBJECT CLASSES

As stated earlier, the basic unit of decomposition in the object-oriented paradigm is the object. Therefore it would be appealing to develop a measure of the complexity of an object. Here the question which arises is "What is it that makes an object more (or less) complex than some other object?" The seemingly reasonable answers to that question which have so far been presented include: the size (volume) of the object, the complexity of the operations (methods), the number of operations, the complexity of the object values (instance variables), the number of object values, and inheritance. Also, there is certainly the possibility of providing a measure based on some combination of these factors.

Since an object is implemented as a set of program code which defines its characteristics and operations, it should be reasonable to measure its size using volume metrics such as Halstead's. Applying Halstead's measures to objects will be discussed in more detail in the next chapter.

Since object operations are commonly implemented as individual functions, it makes sense to compute the complexity of the code defining the operations of an object and then combine the complexities of the operations to determine the complexity of the entire object. Since the McCabe complexity of a module (i.e., the number of basic execution paths

represented by the module) is defined as the sum of the McCabe complexities of its functions, then we should be able to define the McCabe complexity of an object (i.e., the number of basic execution paths represented by the object) as the sum of the McCabe complexities of that object's operations. This methodology will be discussed in Chapter 6.

Intuitively, the more operations defined in 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 any 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 large, 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 reducing its usefulness. Note that this is equivalent to what happens in the functional paradigm; that is, when one function grows too complex it is decomposed into multiple functions. An actual numerical limit on the number of operations is undefined at this point but is thought that the limit 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.

Since object-oriented design is data centered rather than function centered, evaluating an object's complexity based on the complexity of 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 [14], and this is certainly applicable to the object-oriented paradigm.

At first glance, it would seem sensible when considering the complexity of an object to also consider the complexity of the objects which it inherits, particularly since higher-level objects (which intuitively seem more complex) are often built upon several inherited base objects. However, this type of consideration has at least one serious side effect. If we place a limit (say L) on the complexity of any object (regardless of how we are measuring that complexity), and the complexity of a parent object (C_p) is included (cumulatively) into the complexity of its child object (C_c), then we have effectively said that the child object can not have a complexity of more than ($L - C_p$) before the complexity of the inherited-parent object is included. Also, by considering the complexity of inherited objects, we have placed a limit on the amount of inheritance. To see that this is so, imagine that you have placed a limit on object complexity at 100 and you combine the complexities of parent and child objects with addition. Then if you have a base object with a complexity of 99, any object needing to inherit that base object would not be allowed to have a complexity of more than one, and would not be allowed to inherit any other objects (assuming you are allowing multiple inheritance). Also, the newly formed object, having a complexity of 100 already, could not be inherited by any other objects. While it appears

that considering the complexity of inherited objects when computing an object's complexity imposes some serious constraints, considering the number of objects inherited or the level of inheritance could provide some measure of object complexity. For example, an object which inherits five objects might be considered more complex than an object which inherits one object, and an object which inherits no objects (thus having an inheritance level of zero) might be considered less complex than an object which inherits another object which has itself inherited another object (thus having an inheritance level of 2).

Each of the preceding factors of object complexity needs to be studied in considerably more detail. There is a need to develop measures based on these factors and test those measures for correlations relative to error rates, testability, and maintainability. Such effective measures of object complexity could then be used to strengthen the process of object-oriented software development.

5. APPLYING SOFTWARE SCIENCE TO OBJECTS

A tool for computing the length, vocabulary, volume, estimated program level, and estimated programming effort of objects implemented in LISP Flavors has been developed and tested. Table 1 shows its results after analyzing the LISP source code for a set of objects of different personally-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 personally-perceived complexities of the objects.

There are a few noteworthy aspects of applying Software Science to LISP source. Firstly, since the Software Science tool operates on LISP

Table 1: Software Science Measurements of Various Objects

Object	No. of Methods	Length	Vocab.	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
GRAPHIC-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-EDITOR	24	675	193	5124.91	.0184	277789.63

constructs which are read as data in the form of LISP lists, pairs of parentheses are not counted as tokens. Therefore, this LISP expression:

```
(+ A B)
```

is considered as having one operator and two operands, or three tokens total. Secondly, distinguishing accurately between operators and operands in LISP can be very difficult, if not impossible. Any symbol in LISP can represent both a variable and a function at the same time. How it is used is largely dependent on the context and is often dynamically determined at run time. In LISP, just because a token follows an opening parenthesis does not make it a function. Consider for example how local symbols are declared in the LET construct:

```
(LET (A B C) ...)
```

Here, the symbol **A** is an operand along with the symbols **B** and **C**. Also, any function name may be used as an operand in order to access its function definition as data. These are just a few of the complications in separating operators from operands. The Software Science tool developed by the author works on the premise that any token following an opening parenthesis might be an operator. The LISP **FUNCTION** predicate is then used to determine if the token represents a function. This technique is by no means perfect, but does provide a workable compromise. Because of this difficulty in recognizing the operator/operand dichotomy, the accuracy of some of our Software Science measurements could be in question.

The author's Software Science program computes the volume of an object by first computing the length and vocabulary relative to the entire object as opposed to computing and summing the volumes of the individual object components. It is thought that the former provides a better estimate of the object's volume. Since it has been shown that the volume metric is not additive [15], it follows that we will not always get the same results from both techniques. However, we can show that the volume computed relative to the entire object will always be greater than or equal to the sum of the volumes of the object's components.

To prove this, assume that object O has two operations, m1 and m2. Let n_0 be the number of distinct tokens relative to the entire object, n_{m1} be the number of distinct tokens in m1, and n_{m2} be the number of distinct tokens in m2. Likewise, let N_0 be the total tokens in the object, N_{m1} be the total tokens in m1, and N_{m2} be the total tokens in m2.* First we show that $N_0 = N_{m1} + N_{m2}$. Any token in an operation is a token in the object, so $N_{m1} + N_{m2} <= N_0$. Also, any token in the object is in at least one of the operations, thus $N_0 <= N_{m1} + N_{m2}$.

Since the first occurrence of a token in an operation can be taken to be the unique occurrence of the token in both the operation and the object, we can say that $n_{m1} <= n_0$ and $n_{m2} <= n_0$.

---

*For the purposes of argument, we will not count the tokens in the object-class declaration. However, that could be considered the same as adding another operation definition, so the proof will still hold in general.

Thus, we can argue that the sum of the volumes of the operations, V_{o1} and V_{o2}, is less than or equal to the volume of the entire object, V_O, treated as a whole:

$$
\begin{aligned}
V_{o1} + V_{o2} &= N_{o1}\log_2 n_{o1} + N_{o2}\log_2 n_{o2} \\
&= \log_2(n_{o1}\hat{}N_{o1}) + \log_2(n_{o2}\hat{}N_{o2}) \\
&= \log_2[(n_{o1}\hat{}N_{o1})(n_{o2}\hat{}N_{o2})] \\
&<= \log_2[(n_O\hat{}N_{o1})(n_O\hat{}N_{o2})] \\
&= \log_2[n_O\hat{}(N_{o1} + N_{o2})] \\
&= (N_{o1} + N_{o2})\log_2 n_O = N_O\log_2 n_O = V_O
\end{aligned}
$$

By proving that the volume of the object treated as a whole, V_O, is always greater than or equal to the combined volumes of the object components, we can at least consider V_O as an upper-bound on the object's volume. So by using V_O we will not underestimate the object's volume (size).

The program level as defined by Halstead is dependent on the potential volume of a program. As noted earlier, determining potential volume can be very difficult, and the use of objects would seem to multiply the difficulties since an object interface can support several different messages each possibly requiring a different set of parameters. Consequently, Halstead's approximation formula for the program level was applied to the example objects instead of trying to determine the exact program levels. The resulting level estimates are shown in Table 1. Here it is interesting to consider the program level of objects which have *no*

operations. Such an object is commonly known as a *base object*. Base objects are not meant to be instantiated and used separately. Rather, a base object is meant to be inherited by other objects which then possibly provide operations on the base object's data.* Since base objects do not explicitly provide any interface, the potential volume of a base object can be considered undefined. This in effect invalidates the program-level measurement of base objects. Our actual results from computing the program level of objects which define no operations produced levels greater than one. This would imply for a particular object that the object can be implemented using less volume than the least possible amount of volume which could be used to implement the object! This is clearly unreasonable. Therefore, we will consider Halstead's program-level measurement to be undefined for objects which do not define any operations.

Effort measurements listed in Table 1 were computed using the volume measures and estimates of the programming levels of the objects. It can be seen that the effort, as would be expected, increases along with the volume of the object. Since Halstead's effort measurement is based upon the program level, our conclusion that the program level of an object which defines no operations is undefined leads to questions regarding the effort measurement for such base objects. The effort measurements which were computed for objects with no operations (methods) actually seem

---

*An exception might be an object which defines no operations itself but inherits the operations of a parent object. However, the actual usefulness of such an object is questionable.

reasonable and are included in Table 1. However, these values should be considered as suspect pending further analysis. Another noteworthy consideration is the effect of abstraction on programming effort. Two of the example objects (CIRCLE and GRAPHIC-CIRCLE) that were originally combined through inheritance were recoded as one object, named COMBINED-CIRCLE. The original objects each defined some object values and operations (including one operation which was defined by the parent object and then redefined by the child object). It can be seen in Table 2 that the programming effort without the use of abstraction and inheritance is actually higher than the combined effort required for the original objects. This lends support to the intuition that increased abstraction reduces programming effort.

Much more work needs to be done in the area of developing and applying volume metrics to objects. Halstead's measures should be applied

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

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

in a consistent manner to object-oriented systems developed in different languages to determine the accuracy of these measures independently of the software implementation. Also, Halstead's predictors of length, effort, and time should be defined for objects and applied during the object-oriented design process, and then compared with actual programming results. These items are outside the scope of this project and are left to a later study.

6. APPLYING CYCLOMATIC COMPLEXITY TO OBJECTS

We compute the complexity of each object operation (method) as defined by McCabe [16], namely, the number of control-flow predicates plus one. Also, the set of operations of an object potentially represents a collection of algorithms which, as shown by McCabe [17], has a complexity equal to the summation of the complexities of each of the individual algorithms. Recall that the McCabe complexity is the size of the basic set of execution paths of a subroutine. Thus, by computing the McCabe complexity of each operation defined by an object class and then summing the complexities, we have found the size of the basic set of execution paths represented by that object class.* For example, assume an object has two operations, **A** and **B**. The code which defines operation **A** contains four predicates. Thus, operation **A** has a McCabe complexity of five. Likewise, operation **B** contains six predicates, resulting in a McCabe complexity of seven. So by summing the McCabe complexities of all our object's operations, namely **A** and **B**, we find our object has a McCabe complexity of 12. This technique has been applied to objects implemented in LISP Flavors and produced reasonable results. The program used was

*In the interests of simplicity, we are assuming an absence of any in-line methods (like those available in the C++ language) or of any potentially complex instance-variable initialization code (like that which is possible in LISP). Only the explicitly coded operations for an object have been considered.

itself implemented in LISP and was designed to recognize all the Common-LISP structured control-flow constructs.*

The recommended limit on the McCabe complexity of a subroutine is 10. Since object operations are normally implemented as subroutines (functions, procedures, etc.), we should limit their complexity to 10. However, this raises the question of what should be the limit on the McCabe complexity of an object? If we wish to control complexity in the object-oriented paradigm, then we must place a limit on the complexity of an object. This follows logically from the fact that the object is considered to be the basic unit of decomposition. So we must provide some guide as to when an object should be decomposed (or abstracted) into separate objects.

As mentioned earlier, we believe that the complexity of an object and the number of operations allowed for an object should both be considered. By computing the McCabe complexities of the same set of graphics objects which were examined in the previous chapter we see, in Table 3, that the complexity of an object roughly correlates with the number of operations. We believe this to be true in general. This may lead one to conclude that simply limiting the number of operations defined by an object will effectively limit the complexity of that object. However, if we do not limit the complexity of each operation as well then the object's complexity will certainly not be under control. The number of operations may stay the same, but the operations themselves may grow

*The Common-LISP control-flow constructs recognized include **IF, WHEN, UNLESS, COND, TYPECASE, DO,** and **DO***.

increasingly complex! Likewise, if we limit only the complexity of the operations without limiting the number of operations, we have not effectively controlled the object's complexity. The operations themselves may remain manageable, but there may be more and more operations. It is apparent that a method for limiting an object's complexity must take into account both the number of operations and each operation's complexity.

One possibility is to consider the set of algorithms which define the object's operations as a whole, thus removing the need to consider the complexity of each individual operation or even the number of operations. This is equivalent to our application of Halstead's measures in the preceding chapter. However, this technique could not be used to apply McCabe's measure of cyclomatic complexity. The reason being that each of an object's operations effectively represents a separate "entry point" into the object; i.e., a point at which the flow of control within the

Table 3: McCabe Complexities of Various Objects

Objects	No. of Methods	McCabe Complexity
BASIC-GRAPHICS-OBJECT	0	0
CIRCLE	3	3
RECTANGLE	3	3
GRAPHIC-CIRCLE	2	2
GRAPHIC-RECTANGLE	2	2
SIMPLE-COMMAND-PANE	0	0
SIMPLE-GRAPHICS-PANE	0	0
SIMPLE-EPP-GRAPHICS-EDITOR	24	34

object can begin. McCabe's complexity metric is defined for algorithms with only one entry point. This is in fact the reason that we have been computing the McCabe complexity of an object as stated above: the sum of the McCabe complexities of the operations. An object's set of operations represents a set of distinct algorithms and the complexity of each of these algorithms must be computed individually.

An obvious and possibly workable approach to the problem is to limit both the number of object operations and the complexity of each operation. So if we allow only 50 operations and allow each operation a maximum complexity of 10 (the generally-accepted limit for functions), then the maximum complexity of an object would be 500 (50 times 10). However, we believe that 50 operations each with a complexity of 10 is too complex for one object and that defined in this manner, 500, or any other limit derived in the same fashion, does not represent an intuitive indicator of an object's complexity.

A better approach is that of limiting the complexity of each of the operations (to 10) and setting a "reasonable" limit on the complexity of the entire object **without** explicitly limiting the number of operations. In fact, once a complexity limit for an object is decided upon it will in practice implicitly provide a limit for the number of operations as well. This limit on the number of operations will vary inversely with the combined complexities of the operations. For example, if we limit the McCabe complexity of an object to 100, then we may have 10 operations each with a complexity of 10, or 100 operations each with a complexity of one. This approach not only limits the complexity of the operations and the

number of operations, but relates the two as well. The effort required to test an object with 10 operations each having a complexity of 10 should be equivalent to the effort required for an object with 100 operations each having a complexity of one. Unfortunately, this approach also leads us back to the problem of finding a reasonable limit for the McCabe complexity of an object.

An argument supporting 100 as the limit for an object's McCabe complexity is that one can think of the object's interface (set of operations) as resembling a set of cases as in a "case" statement. As such the interface should be subject to the same complexity limit as a normal function, namely 10. Then 10 allowed operations each with a complexity of 10 gives a limit of 100. However, with less complex operations we would allow the number of "cases" (operations) to increase, but still limit the total object complexity to 100. This suggested limit of 100 seems intuitively reasonable and data collected relating the McCabe complexities of objects with the number of known defects in those objects lends favorable support to it [18]. However, there is certainly a need for more empirical evidence and it is hoped that future research into the degree of correlation between the error rates of object-oriented code and the McCabe complexity of objects will point towards an optimal complexity limit.

7. SUMMARY

Research into both the areas of software metrics and object-oriented programming has intensified during the past decade. However, there is an apparent lack of intensive study concerning the application of software metrics within the object-oriented paradigm. Here we have discussed some of the factors affecting the complexity of software objects and the potential application of Halstead's and McCabe's metrics to measure the size and complexity of an object. The concern here has not been to recommend any metric over the other, but rather to identify some likely means of applying the metrics towards objects. Metric tools developed by the author for use in evaluating objects seemed to produce reasonable results, though the accuracy of the Software Science measures may be questionable.

Potentially important factors affecting the complexity of object-oriented software include: the size (volume) of the object, the complexity of the operations, the number of operations, the complexity of the object values (instance variables), the number of object values, and inheritance. The degree to which each of these factors affects the complexity of objects has yet to be determined experimentally. It can be seen in advance though that factoring the complexity of inherited objects into the complexity of the inheriting object has the serious side effect of limiting inheritance.

We have shown that Halstead's volume and effort measures roughly correlate with the sizes and complexities of objects as perceived by the author. Also, the use of abstraction and inheritance may reduce programming effort. Halstead's measurement of the program level is effectively undefined for objects which define no operations (base objects). Therefore, effort measurements (which are theoretically based upon the program level) of base objects are questionable as well. The accuracy of Halstead's predictors of program length, programming time, and programming effort relative to object-oriented programming needs further study.

By computing and summing the McCabe complexities of an object's operations, we can compute the McCabe complexity of an object. We feel that a reasonable approach to controlling object complexity is to set an independent limit for the complexity of an object which will in turn implicitly limit the number of operations. It is generally agreed that a McCabe complexity of 10 is a reasonable limit for a subroutine. We suggest a limit of 100 as a reasonable McCabe complexity of an object. However, further empirical studies are needed to validate this claim.

During the object-oriented design process 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 object-oriented programming does not by itself provide for the management of complexity. Object complexity must be measured and effectively limited before software complexity can be controlled in the object-oriented paradigm.

WORKS CITED

1. Kearney, Joseph K., Robert L. Sedlmeyer, William B. Thompson, Michael A. Gray, and Michael A. Adler. 1986. Software Complexity Measurement. Communications of the ACM, Vol. 29, No. 11 (November): 1044.

2. Siyan, Karanjit S. 1989. Coping with Complex Programs. Dr. Dobb's Journal (March): 60.

3. Ibid., 65.

4. Halstead, Maurice. 1977. Elements of Software Science. New York: Elsevier North-Holland, Inc.

5. Fitzsimmons, Ann and Tom Love. 1978. A Review and Evaluation of Software Science. Computing Surveys, Vol. 10, No. 1 (March): 3.

6. Levitin, Anany V. 1986. How to Measure Software Size, and How not to. Proceedings of the IEEE COMPSAC'86: 316.

7. McCabe, Thomas J. 1976. A Complexity Measure. IEEE Transactions on Software Engineering, Vol. SE-2, No. 12 (December): 308.

8. Ibid., 314.

9. Walsh, Thomas J. 1979. A Software Reliability Study Using a Complexity Measure. In Proceedings of the National Computer Conference. New York: AFIPS, 766.

10. Harrison, Warren A. and Kenneth I. Magel. 1981. A Complexity Measure Based on Nesting Level. In ACM SigPlan Notices (March): 64.

11. Ramamurthy, Bina and Austin Melton. 1986. A Synthesis of Software Science Metrics and the Cyclomatic Number. Proceedings of the IEEE COMPSAC'86: 310.

12. Evangelist, Michael. 1984. An Analysis of Control Flow. Proceedings of the IEEE COMPSAC'84: 390.

13. Wirfs-Brock, Rebecca and Brian Wilkerson. 1989. Object-Oriented Design: A Responsibility-Driven Approach. OOPSLA'89 Proceedings (October): 71.

14. Tsai, W. T., M. A. Lopez, V. Rodriguez, and D. Volovik. 1986. An Approach to Measuring Data Structure Complexity. Proceedings of the IEEE COMPSAC'86: 240.

15. Levitin, 317.

16. McCabe, 314.

17. Ibid.

18. Fiedler, Steven P. 1989. Object-Oriented Unit Testing. Hewlett-Packard Journal (April): 73-74.