# A Software Complexity Metric for C++

Al Lake
Curtis Cook
Department of Computer Science
Oregon State University
Corvallis, Oregon 97331-3202

92-60-03

# A Software Complexity Metric for C++

Al Lake
Curtis Cook
Department of Computer Science
Oregon State University
Corvallis, Oregon 97331-3202
(503) 737-5564
lake@cs.orst.edu
cook@cs.orst.edu

## 1. Introduction

Object-oriented programming languages have been promoted as the solution to the size and complexity issues of program development and maintenance [Cox86, and Poun90]. Object-oriented programming (OOP) is purported to provide a more direct way of modeling the world than action-oriented (procedural) programming [Wegn87]. Object-oriented programming has different components [Cox86], different methods of design [Blai89, Cox86, and Wegn87], and different methods of testing [Fied89, and Perr90] than imperative programming languages.

There has been little investigation of the quality and complexity of object-oriented programs. Object-oriented programming is so new that there are few accepted principles and guidelines. We do not know what makes an object-oriented program difficult or easy to understand, test, or maintain. The long-term goal of this research is to develop a quantitative method to determine the areas of greatest complexity in object-oriented programs, thus enabling a programmer to focus on the most error-prone or difficult to understand portions of code.

In this paper we describe an object-oriented software complexity metric and our preliminary efforts in validating the metric. Our basic premise was that because of the function orientation and separation of procedures and data in traditional procedural programming, traditional software complexity metrics (e.g. Lines of Code, McCabe's $v(G)$ [McCa76], and Halstead's Software Science [Hals77]) do not adequately measure the complexity of object-oriented programs. Our approach was to identify the OOP problem areas in the current literature and where appropriate adapt procedural metrics or develop new metrics that model these problem areas. Because extensive searching of the inheritance tree was common to all of the problem areas and because of the crucial role of classes in object-oriented programming, for our metric we concentrated on measures of the class inheritance tree. Our initial hypothesis is that the depth and size features of the inheritance tree are major contributors to complexity.

Several papers have proposed metrics similar to ours, but have not conducted empirical studies to validate them. Chidamber and Kemerer [Chid91] proposed a suite of metrics for object-oriented design: weighted methods per class, depth of inheritance tree, number of children, coupling between objects, all methods available to a class, and lack of cohesion in methods. They did not attempt to empirically evaluate their metrics. Instead they formally evaluated their metrics against Weyuker's metric evaluation property list [Weyu88]. Morris [Morr89] defined nine candidate metrics: methods per object class, inheritance

dependencies, degree of coupling between objects, degree of cohesion of objects, object library effectiveness, factoring effectiveness, degree of reuse of inheritable methods, average method complexity, and application granularity. Morris did not attempt to empirically evaluate the complexity metrics proposed. He mapped the candidate metrics to productivity impact variables proposed by Booch [Booc86] and Seidewitz and Stark [Seid86] and he subjectively judged the influence of these metrics on impact variables: maintainability, reusability, extensibility, testability, comprehensibility, reliability, authorability. In another related paper Coppick and Cheatham [Copp92] applied McCabe's $v(G)$ and Halstead's software science to objects in Lisp Flavors. They constructed a tool and computed metrics for part of a simple graphics editor program. Based on the limited data they suggested an upper limit for $v(G)$ for an object, but stressed the need for more empirical data.

To test our ideas we developed a software complexity metric for C++. We developed a program that computes a wide variety of size and inheritance tree measures from C++ source code. This paper reports our initial evaluation of the metrics. Our preliminary results support our hypothesis that the size and depth features of the inheritance tree are indicators of complexity in object-oriented programs.

This paper is organized as follows. In the next section, we describe the essential features of OOP, the basic OOP definitions and factors, and the OOP problem areas. The third section defines the basics of the traditional classes of software complexity metrics, and presents our conclusions for use of these classes in OOP. In the fourth section we define our assumptions, the reasons for picking C++, the components of our C++ metric, and the unique method we use to augment our software complexity metric tool. In the fifth section we give our preliminary findings, and the similarities and differences in the C++ code we have analyzed during testing of the metrics tool. Finally in the last section we give a description of a preliminary experiment we conducted.

## 2. Object-oriented Programming Background

Before presenting OOP complexity issues we will first fully characterize the essential features of OOP. Since there is no single accepted definition for OOP, we will give what are considered by most authors to be the essential ingredients and factors of OOP.

### Object-oriented Programming Definitions

An object-oriented system should have the following [Blai89]:

a. Possess encapsulation and inheritance,
b. Have set-based abstraction, and
c. Support inclusion polymorphism and operation polymorphism.

Most authors agree that the two most important features of an object-oriented programming language are inheritance and encapsulation [Budd90]. Inheritance and encapsulation are defined as [Budd91]:

Inheritance - A class may be defined as an extension or restriction of another. All the information known about one class can be inherited by a subclass. Classes can be ordered hierarchically with subclasses inheriting behavior from superclasses. This allows a programmer to structure a solution by building on a base of existing code.

Encapsulation - An object is encapsulated if it incorporates an operation set and a data set into a single entity. So encapsulation provides objects with both data and operations to perform the requested activities. This allows the programmer to hide implementation details and develop a solution in terms of high level abstractions by incorporating actions (operations) with data.

Since inheritance is the mechanism of deriving or defining a new class from an old one, the description of a derived class is inherited from the base class. This description can be altered by adding members, overloading existing member functions, and modifying access privileges [Pohl89], producing a complex tree hierarchy when all the class interrelationships are juxtaposed.

**Object-oriented Problem Areas**

Our literature review showed the following OOP problem areas:

Classes and inheritance [Knud89] - the base class defines the action of the derived class through inheritance, but the inheritance tree must be searched for the properties of the base class.

Operator overloading [Wirf90] - operator overloading for class operands allows the implementor of a class to define the semantics of each operator separately. Operator overloading occurs when variables of the same name with similar or totally different functionality are defined in the same inheritance tree. Overloading requires a context sensitive search of the inheritance tree to find the base class and determine the attributes of the operator.

Encapsulation abstraction [Wirf90] - encapsulation is the enforcement of the abstraction barrier by hiding the implementation details from the externally available operations and functions. Encapsulation allows information hiding to take place, which requires a search of the inheritance tree.

Constructors and destructors [Knud89] - permit dynamic allocation of space during the execution of the program. Derived classes can have private areas that require knowledge of the base class prior to programming. This knowledge requires a lookup of the base class definition to determine private and public attributes, class initialization, class deallocation, and dynamic memory allocation.

Yo-yo problem [Taen89] - with software reuse, construction and inheritance, an interclass dependency problem can occur. For example, any time a class calls itself or passes itself as an argument, a message is sent to the base class of the calling object. The base class interprets the message. A lookup of the calling class must be done, which is like a yoyo going down to the bottom of its string. If the base class does not implement a function for the message, a search of the superclass hierarchy chain occurs looking for the class that does implement the function. This is like the yoyo going back up the string [Taen89]. The yo-yo problem necessitates the lookup of the base class to determine behavior and data that activates a search of the inheritance tree.

Polymorphism [Kors90] - the ability of procedures or functions to operate on more than one type. In object-oriented languages the ability to create a polymorphic function is due to message passing and inheritance. Since an object will determine which polymorphic function to utilize by characteristics that are inherited there must be a lookup of the inheritance tree to determine which function is the appropriate one. This lookup can be particularly difficult for the programmer because there can be multiple functions with the same function name and the variables in the function call need to be analyzed to help determine the proper function utilized.

It is important to notice that extensive searching of the inheritance tree is common to all of these object-oriented problem areas. All uses of class behavior require a lookup, i.e., search, of the inheritance tree to determine the proper behavior and characteristics to be attributed to the class.

## 3. Software Complexity Metrics

### Definition of source code complexity

Software complexity metrics are objective measures of how complex source code is and how difficult it may be for a programmer to test, maintain, or understand programming source code [Cook84]. Software complexity metrics do not measure the complexity itself, but instead measure the degree to which those characteristics thought to contribute to complexity exist within the source code [Oman90]. For example, if a program has complex control flow with many logical paths through the code, the program is thought to be difficult to test and more likely to have errors. Hence, a software complexity metric for this example of complex control flow is the number of decision statements.

### Traditional Classes of Software Complexity Metrics

The classes of software complexity metrics for traditional procedural languages are:

Size metric - a measure of the size of the source program. Examples of this metric include counts of lines of code, tokens, functions, and adapted or modified lines of code [Jone78 and Duns84].

Data structure/Information flow - a count of the amount of data input to, processed in, and output from a data structure metric. Examples of this metric include variable count, live variables count, variable span, fan-in and fan-out of the module, and global variable count [Cont86].

Logic structure - a measure of the control flow or logic execution of a program. Examples of this metric include decision count, McCabe's cyclomatic number, and nesting level [McCa76].

Logic structure complexity metrics are considered the least applicable to inheritable features because there are no control flow issues in true OOP. Hence we will only include size and information flow metrics in our OOP metric.

## 4. Software Complexity Metric for C++

Our software complexity metric for object-oriented programs was influenced by the problems for object-oriented programming that have been identified in the literature and by traditional software complexity metrics. Recall that extensive searching of the inheritance tree was common to all six OOP problems. Size and information flow are the traditional software complexity metrics incorporated into our metric. Control flow was not included because control flow issues do not impact the inheritance tree. Our metric measures the depth, size, and amount of information passed up and down the inheritance tree.

We choose to test our ideas by developing a software complexity metric for the programming language C++. We choose C++ because a large amount of C++ code exists and there is a large amount of programming currently being done in C++. Since C++ is an extension of C using C++ will also allow us to evaluate the effectiveness traditional complexity software metrics on object-oriented programs.

**Components of CPPOOM**

CPPOOM (C++ Object-oriented Metric) is written in C++ and consists of the following four programs:

Includes - returns counts of the number of user-defined include files, called local files, and the number of global library files. This program creates a temporary file, CPPOOM.TMP, which includes all local include files expanded. An optional display shows all of the local include files with counts in physical order.

Functions - returns a count of the number of procedural functions. An optional display shows all of the function names with counts in physical order.

Operators/Operands - returns counts of the number of tokens, keywords and library functions. An optional display shows the complete list of C++ keywords and functions with counts.

Inheritance Tree - parses the temporary file created by the Includes program, CPPOOM.TMP, to create the complete inheritance tree and returns counts of number of classes, class depth, number of lines of code in classes, number of subclasses, number of polymorphic member functions, number of overloaded functions, number of member functions, and number of data variables.

Figure 1 is an example of the output from our metrics tool (CPPOOM) for a sample program [Budd91]. In the right hand column we added an annotated description of what the metrics values represent.

```
CPPOOM version 0.2
Software Complexity Metric for C++                          Date:  9/10/91
File: expr.cc

Total number of includes:                    4      counts the number of user-defined local
  Number of local includes:                  3      includes and library includes
  Number of library includes:                1

Number of functions:                         4      counts the number of procedural functions

Number of comment lines:                    23
Total number of lines:                     269      counts the number of lines in the file
Total nonblank lines (SLOC):               231
Total blank lines:                          38

Number of C++ tokens:                      145
Total Number C++ keywords:                 137      counts the number of C++ keywords and functions
Distinct C++ keywords:                      11
Total Number of C++ library functions:       8
Distinct C++ library functions:              2

Inheritance Counts With All Include Files Expanded
Class Counts:
  Total number of classes:                  57      number of classes in the inheritance tree
  Total base classes:                        3      number of root classes in the tree
  Average class depth:                    3.02      average class depth
  Maximum class depth:                       4      deepest class depth
  Average class LOC:                      7.88      average class lines of code
  Maximum class LOC:                        27      largest class lines of code
  Minimum class LOC:                         4      smallest class lines of code
  Greatest class uses:                       0      greatest number of class uses
  Maximum subclasses:                       24      greatest number of subclasses
  Class references outside tree:             1      shows the number of references beyond the tree
  Total # of lines (expanded):             848      total lines expanded
  Total SLOC (expanded):                   729      total source lines of code expanded
  Total lines in classes:                  449      total source lines of code in classes
  Procedural lines in classes:               0      total procedural lines in classes
  Total polymorphism count:                202      total number of polymorphic member functions
  Maximum polymorphism count:               32      largest number of polymorphic member functions
  Total overload count:                    351      total number of overloaded member functions
  Maximum overload count:                    9      largest number of overloaded member functions

Member function counts:
  Total member functions:                  194      total number of member functions in the tree
  Maximum member functions:                 17      largest number of member functions in a class

Data variables counts:
  Total data variables:                     71      total number of data variables in the tree
  Maximum data variables:                    4      largest number of data variables in a class
```

**Figure 1. CPPOOM Output**

## Optional Displays of CPPOOM

CPPOOM has several optional displays that provide more detailed information. These optional displays include the following:

a. the class hierarchy list, shown in Figure 2.,
b. the number of nodes at each level of the inheritance tree,
c. the member functions display, which shows the class depth of each member function and a count of the number of polymorphic objects,
d. the names and sizes of the local include files,
e. the operator and operands, a list of C++ keywords and C++ library functions and the number of uses, and
f. the procedural function names and sizes.

The class hierarchy display lists the extended class hierarchy by indenting to depict the inheritance tree relationship of each class. The display contains the class name, the class type, the number of lines of code, class depth, the number of immediate subclasses, the number of public member functions, and the number of public data variables. Additionally, the names of the member functions of each class can also be listed below each class. Figure 2 is a partial listing of the class hierarchy for a sample program [Budd91].

| Class Name | Class Type | LOC | Depth | Subclass | Funcs | Variables |
|---|---|---|---|---|---|---|
| expression | undefined | 27 | 0 | 8 | 17 | 2 |
| variable | expression | 17 | 1 | 4 | 8 | 1 |
| rankVariable | variable | 4 | 2 | 0 | 1 | 1 |
| shapeVariable | variable | 4 | 2 | 0 | 1 | 1 |
| valuesVariable | variable | 4 | 2 | 0 | 1 | 1 |
| temporary | variable | 9 | 2 | 0 | 3 | 1 |
| scalar | expression | 16 | 1 | 0 | 9 | 1 |
| rscalar | expression | 8 | 1 | 0 | 3 | 1 |
| condexpr | expression | 11 | 1 | 0 | 4 | 1 |
| searchexpr | expression | 11 | 1 | 0 | 3 | 1 |
| letexpr | expression | 25 | 1 | 4 | 9 | 4 |
| letSigma | letexpr | 16 | 2 | 0 | 6 | 1 |
| letCollect | letexpr | 19 | 2 | 2 | 7 | 1 |
| letCompress | letCollect | 11 | 3 | 0 | 4 | 1 |
| letExpand | letCollect | 10 | 3 | 0 | 4 | 1 |
| letVar | letexpr | 11 | 2 | 1 | 4 | 1 |
| letSort | letVar | 8 | 3 | 0 | 3 | 1 |

**Figure 2. Class Hierarchy - Optional Display**

The following terms are used in Figure 2:

Class Type is the type of the base class.
LOC are the total lines of code count of the class being defined.
Depth is the depth of this class in the inheritance tree.
Subclass is the count of immediate subclasses.
Funcs is the count of the public member functions defined within this class.
Variables is the count of the public data variables within this class.

**Unique Approaches of CPPOOM**

Our metric utilizes several unique approaches to analyzing object-oriented programs. First, CPPOOM differentiates between member functions and procedural functions. Member functions are a part of the inheritance tree complexity and procedural functions are part of the procedural complexity. Member functions have a different use than procedural functions. A member function is declared and allows the base class or a derived class to have particular functions act on its private representation; in other words the privacy of a member function allows the class type to be hidden from all other classes. Procedural functions are dependent upon which path of the program is taken for execution. So that member functions are an inheritable attribute that is a component of the data structure and procedural functions are part of the logic structure. This is an important distinction for C++ because not all object-oriented programming languages have logic structure complexity attributes, i.e., control flow.

Second, CPPOOM analyzes the inheritance characteristics of the entire inheritance tree by expanding all include files prior to computing the various counts. Most object-oriented programmers utilize include files to compartmentalize or abstract the information. This practice allows the programmer to hide the implementation details and develop a solution in terms of a form of high level abstraction. This high level abstraction is another form of

data abstraction, allowing the programmer to not have to consider all of the details of the program.

Finally, CPPOOM analyzes only the public inheritable characteristics of classes, since private characteristics will be restricted to only a few categories of functions. The private elements in each class are not given the same level of importance or significance as the public elements. Private characteristics are characteristics that will be restricted to only a few categories of functions so the significance of these characteristics will be localized. Public characteristics of a class can be considered to be global in nature to the subclasses inheriting. This inheritance obviously depends on the level of the class in the inheritance tree. A derived or subclass inherits the public members of the base class so the private members will not have an impact on the subclasses of the base class.

## 5. Preliminary Validation of Metric

### Preliminary Data Analysis

Traditional software complexity metrics, such as lines of code (LOC) and McCabe's cyclomatic number ($v(G)$), have been used to identify error-prone modules (e.g. the few modules that contain most of the errors), difficult to test modules, and hard to understand modules, to assist in the allocation of resources (e.g. test resources), and to provide reasonable predictions of the number of errors in modules, [Shen85 and Kafu87]. These relationships were validated by our analysis of metrics and performance data (e.g. number of errors, testing time, etc). We feel that the object-oriented metrics should have similar uses, but the same validation steps (analysis of metrics and performance data) needs to be done.

We have applied CPPOOM to several reasonably sized C++ programs (APL compiler, Borland C++ library, C++ instructional code, and an accounts receivable application). Our initial results are that most C++ programmers use small sets of autonomous classes, i.e. separate, unconnected, and shallow inheritance trees. The inheritance trees tended to be either flat or narrow. Most of the programs had fewer than 20 classes of small or moderate size and an inheritance tree of depth one. Of the larger, more complex systems we analyzed, the average size of the classes in these programs was less than 13 lines of code. Inheritance trees of depth three or more were usually very narrow, sometimes almost becoming a linear graph. As we expected, programs with large or deep trees were by far the most difficult to understand.

Our preliminary findings also show that traditional software complexity metrics do not seem to measure object-oriented complexity. Figure 3 gives the metric values for some of the programs we studied. The compiler program, expr.cc [Budd91], is considerably more difficult to understand than the other programs because of the sheer size and depth of it's inheritance trees. Notice that, according to the traditional metrics lines of code (SLOC) and cyclomatic number ($v(G)$), it appears to be as complex or less complex than the other programs. Both todolist.cpp and tododlgs.cpp have over three times as many lines of code. The $v(G)$ of todolist.cpp is nearly twice that of expr.cc. Also, expr.cc has more classes (Total # classes) of smaller average size (Average class size) at a greater class depth (Average class depth) and fewer inheritance trees (# Trees).

| Program Name | expr.cc | todolist.cpp | tododlgs.cpp |
|---|---|---|---|
| Average class depth | 3.02 | 2.56 | 2.47 |
| Average class size | 7.88 | 31.83 | 30.52 |
| Total # classes | 57 | 18 | 19 |
| # Trees | 3 | 6 | 7 |
| SLOC | 848 | 2,896 | 2,749 |
| Total $v(G)$ | 58 | 92 | 50 |
| # Functions | 38 | 38 | 20 |

**Figure 3. Table of Metric Values for 4 Programs**

## 6. Preliminary Depth Threshold Experiment

We conducted a three part experiment to further test our preliminary findings about how the depth of classes in the inheritance tree impacts task performance. We wanted to know if programmers have more difficulty working with classes deeper in the inheritance tree than working with shallow classes. Our hypothesis was that for C++ programs programmers perform common programming tasks more effectively on classes at or near the root of the inheritance tree than on classes at or near the leaves of the inheritance tree. The three programming tasks we selected for this experiment are: program understanding, debugging, and program modification.

### Subjects

Our 11 subjects were graduate students in computer science who were proficient in object-oriented programming and C++. They were selected from 54 students who responded to an e-mail request for subjects. The selection criterion was proficiency in C++ as measured by completion of a graduate level object-oriented programming course and a term project. The subjects were paid $25 for participating.

The subjects averaged 1.7 years in graduate school, 6.0 years of programming, and 2.5 years of professional programming. Four of the subjects had no professional programming experience. Five of the subjects had undergraduate computer science degrees. C++ proficiency ratings were self reported on a 1-10 scale with 10 being the most expert. The average C++ proficiency rating was 7.0.

### Method

The experiment was a within subjects experiment with three tasks: program comprehension, debugging, and modification. A different program was used for each task. We constructed functionally equivalent deep and shallow versions of each program. In the deep version the key class (the class involved in the programming task) was located at the maximum depth or leaf in the inheritance tree. In the shallow version the key class was located at depth zero or the root level. At this level the key class has no inheritable attributes.

We randomly divided the 11 subjects in two groups: A (6 subjects) and B (5 subjects). There were no significant background differences between the two groups. Group A performed the comprehension task on the deep version, the debugging task on the shallow

version, and the modification task on the deep version. Group B did the opposite. They performed the comprehension task on the shallow version, the debugging task on the deep version, and the modification task on the shallow version.

**Materials**

We selected three programs from the Borland C++ Library [Borl91]. The original programs are the deep versions. To create the shallow versions we modified each of the programs to move the key class to depth one. The functionality of the superclasses in the original tree were compressed into this single class so that the original program and the modified program have the same functionality. The modified programs have the same number of classes and functions as the original program, but the maximum class depth of the inheritance tree was decreased by one. Figure 4 gives the size and class depth information for each version of the six programs. Note that the shallow versions have a larger LOC (lines of code) count because of the need to explicitly include the inheritable attributes in the key class definition. The number of classes and functions in the inheritance tree was the same for the deep and shallow versions.

| Program Number | 1 | | 2 | | 3 | |
|---|---|---|---|---|---|---|
| Task | Comprehension | | Debugging | | Modification | |
| Program Version | deep | shallow | deep | shallow | deep | shallow |
| LOC | 315 | 500 | 152 | 186 | 170 | 227 |
| SLOC | 268 | 436 | 128 | 158 | 144 | 193 |
| # classes | 11 | 11 | 5 | 5 | 4 | 4 |
| avg class LOC | 10.09 | 11.09 | 8.20 | 8.75 | 10.33 | 10.54 |
| # functions | 6 | 6 | 5 | 5 | 2 | 2 |
| maximum depth | 5 | 4 | 3 | 2 | 4 | 3 |

**Figure 4. Table of Metric Values for Test Programs**

The source code listings were printed on 8.5" by 11" standard copier paper in Courier 12-pitch type using a laser printer. No bold typeface or other accents were used in the program listings. No comments were included in the source code, and function and variable names were not changed from the Borland code. All of the local include files were incorporated into the single program file and the class definitions were done in the same order.

For the comprehension quiz, the four forward and backward reasoning questions concerned program output and input. The same questions were used for both the shallow and deep versions of program 1. Forward and backward questions were alternated in the quiz. The order of forward and backward questions were alternated within each group of subjects.

The compiler error for the debugging task on program 2 was:

```
line 109: 'IsVisible()' is not a member of 'Circle'
```

Since the error referred to the same class in both the deep and shallow versions, the only difference in the error message for both versions was the line number. The error could be fixed by a simple change to the key class.

The third task was to modify program 3 to add the capability to output the value of the radius of a circle. For the modification only the key class needed to be changed.

**Procedure**

The subjects were given 10 minutes to read and sign the informed consent form, read the general instructions for the experiment, and complete the background questionnaire, (Appendix A). Their first task was a program comprehension task. They were given a program listing and a comprehension quiz with 4 questions (2 forward reasoning questions and 2 backward reasoning questions). They were given 20 minutes to answer the questions. The second task was debugging a program with a compiler error message. Given the program listing and error message identifying the line in error, the subjects had 10 minutes to explain why the error occurred and correct the error. The last task was a program modification task. Given a program listing and the desired modification, the subjects were given 15 minutes to identify where the program should be changed and to indicate the changes. They were instructed to make maximum use of inheritable attributes. A class table of contents, showing the class name within the inheritance tree hierarchy and the class type, was included with each program listing. No other information was included with the class listing. An example of the class table of contents is shown in Figure 5.

**Class Listing**
File: prog1.cpp

| Class Name | Class Type |
|---|---|
| Location | undefined |
| GMessage | Location |
| Point | Location |
| Distance | Point |
| Circle | Distance |
| Arc | Circle |
| Square | Distance |
| OrderedList | undefined |
| Stack | OrderedList |
| Cube | undefined |
| List | undefined |

**Figure 5. Class Table of Contents**

When the exercise was completed, the subjects filled out the Conclusions sheet, that asked for their level of confidence with their work, their perception of the usefulness of the inheritance tree table of contents, and any general comments about the exercise.

**Results**

The times and scores for each group and tasks are given in Figure 6. Values with significance ($p < 0.05$) are shown with two asterisks (**), and values with significance ($p < 0.10$) are shown with one asterisk (*) in Figure 6. The actual scores and times for each subject is given in Appendix B.

Three subjects stated that the time for the comprehension task was not sufficient. They mentioned difficulty with the coding style and one subject thought the task was a "warm up" session. Most of the subjects used the entire 20 minutes as they spent time checking their answers. Since each answer required a sequence of numbers, correctness was scored on a

four point scale: 0 if no answer was attempted, 1 if the answer was wrong, 2 if input or output numbers were correct but in the wrong order, and 3 if input and output numbers were correct and in the correct order. The results are given in Figure 6. A t-test showed no significant differences in the time, number of questions answered, or number of correct answers between the deep and shallow versions. We were surprised because we expected the subjects with the shallow version to take less time and be more accurate. We suspect that the performance similarity may be due to the relative complexity of the class hierarchy depth of 5 in the deep version being compressed into a single class.

Results for the debugging task are shown in Figure 6. The error identification was scored as 0 (incorrect) or 1 (correct). The error correction was scored on a four point scale: 0 if no correction was attempted, 1 if a correction was attempted but was in the wrong place, 2 if a correction was attempted in the right place, but will generate an error, and 3 if the correction is correct. Subjects with the deep version took longer, but not significantly longer. Subjects with the shallow versions were significantly more successful in error identification and scored much higher, but not significantly higher on the correction subtasks.

Results for the modification task was scored much like the debugging task. The error identification was scored as 0 (incorrect) or 1 (correct). The modification change was scored on a four point scale: 0 if no change was attempted, 1 if a change was attempted but was in the wrong place, 2 if a change was attempted in the right place, but will generate an error, and 3 if the change is correct. Syntactical correctness was not a factor in grading this task. Subjects with the deep version spent significantly less time making the modification. However, the subjects with the shallow version were significantly more accurate. This may suggest that the modification task was more difficult than the group with the deep version realized.

| | Group A Mean (Std) | Group B Mean (Std) | Overall Mean (Std) | t-value |
|---|---|---|---|---|
| **Comprehension Task** | | | | |
| | deep | shallow | | |
| time | 17.17 (3.93) | 18.20 (3.12) | 17.64 (3.62) | 0.4302 |
| # answered | 3.67 (0.75) | 3.80 (0.40) | 3.73 (0.62) | 0.3249 |
| correctness | 2.33 (0.47) | 2.40 (0.49) | 2.36 (0.48) | 0.2075 |
| **Debugging Task** | | | | |
| | shallow | deep | | |
| time | 4.08 (2.71) | 7.00 (3.35) | 5.41 (3.35) | 1.4435 |
| identification | 1.00 (0.00) | 0.40 (0.49) | 0.73 (0.45) | 2.7136** |
| correction | 2.50 (0.50) | 1.20 (1.17) | 1.91 (1.08) | 2.2356* |
| **Modification Task** | | | | |
| | deep | shallow | | |
| time | 12.00 (2.16) | 14.60 (0.80) | 13.18 (2.12) | 2.3061** |
| identification | 0.50 (0.50) | 1.00 (0.00) | 0.73 (0.45) | 2.0226* |
| change | 1.67 (0.75) | 2.80 (0.40) | 2.18 (0.83) | 2.7618** |

**Figure 6. Table of Tasking Order for Test Programs**

A class table of contents listing was included with each program listing. Ten of the 11 subjects drew some type of graph sketch of the class hierarchy as they studied the program

listing. The one subject who did not draw the inheritance tree structure was the only subject who felt the class table of contents was not useful; however, the subject did state that the class table of contents listing might be useful in a larger program.

At the end of the experiment we asked the subjects to rate on a scale from 0 to 5 (with 5 as most confident) their overall confidence in their answers. There was little difference between the mean of the ratings for subjects in Group A and in Group B as shown in Figure 7. There was no significant correlation between the rating and task performance. The two subjects with the highest rating did not have the highest scores on all of the tasks and the one subject with the lowest rating did not have the lowest scores on all of the tasks. Generally, the confidence ratings and comments of the students shows a lack of confidence in their work. This may be due to the fact that the subjects could not execute the programs to verify the error was corrected or the modification worked. Three of the subjects stated that the time for the comprehension task was not sufficient to complete the task successfully. The subjects level of confidence in their overall work, their perception of the use of the inheritance class table of contents and the usefulness of the class table of contents is shown in Figure 7. There was not a significant difference in programmer confidence between Group A and Group B.

| | Group A Mean (Std) | Group B Mean (Std) | Overall Mean (Std) |
|---|---|---|---|
| Confidence | 2.83 (0.69) | 2.60 (1.02) | 2.73 (0.86) |
| TOC Use | 0.67 (0.47) | 1.00 (0.00) | 0.82 (0.39) |
| TOC Usefulness | 0.83 (0.37) | 1.00 (0.00) | 0.91 (0.29) |

**Figure 7. Table of Confidence and TOC Use**

## Conclusions

This preliminary experiment indicates that programmers more effectively debug and modify classes at or near the root of the class hierarchy than classes deep in the class hierarchy. The results of this experiment also appear to indicate that there is a point of accumulated complexity where the relative class size and loss of data abstraction make the task of comprehending a single class at the root as difficult as searching through the inheritance tree.

Our results suggest that the depth of the inheritance tree does affect programming tasks. However, more studies will need to be done to determine if there is a depth threshold for programming tasks and the trade-offs between depth and breadth of the inheritance tree.

**References**


[Blai89] G. Blair, J. Gallagher, and J. Malik, "Genericity vs Inheritance vs Delegation vs Conformance vs ...", *Journal of Object-Oriented Programming*, Vol. 2 (3), September/October 1989, pp. 11-17.

[Borl91] *Borland C++*, Version 2.0, Borland International, Incorporated, 1991.

[Booc86] G. Booch, "Object-Oriented Development", *IEEE Transactions on Software Engineering*, Vol SE-12 (2), February 1986, pp. 211-221.

[Budd90] T. Budd, "What is Object Oriented Programming?", *Beaver Bytes Summer, 1990*, Oregon State University Department of Computer Science, 1990.

[Budd91] T. Budd, *An Introduction to Object Oriented Programming*, Addison-Wesley, Reading, Massachusetts, 1991.

[Chid91] S. Chidamber, and C. Kemerer, "Towards a Metrics Suite for Object Oriented Design", *Object Oriented Programming Systems, Languages and Applications (OOPSLA)*, Vol 10, 1991, pp. 197-211.

[Cont86] S. Conte, H. Dunsmore, and V. Shen, *Software Engineering Metrics and Models*, Benjamin/Cummings Publishing Company, 1986.

[Copp92] J. Coppick, and T. Cheatham, "Software Metrics for Object-Oriented Systems", to be presented at the *20th Annual Computer Science Conference*, March, 1992.

[Cook84] C. Cook, "Software Complexity Measure", *Proc. 1984 Pacific Northwest Software Quality Conference*, Portland, Oregon, 1984, pp. 343-363.

[Cox86] B. Cox, *Object-Oriented Programming: An Evolutionary Approach*, Addison-Wesley, 1986.

[Duns84] H. Dunsmore, "Software Metrics: An Overview of an Evolving Methodology", *Information Processing & Management*, Vol 20 (1-2), 1984, pp. 183-192.

[Fied89] S. Fiedler, "Object-Oriented Unit Testing", *HP Journal*, Vol 36 (4), April 1989.

[Hals77] M. Halstead, *Elements of Software Science*, Elsevier North-Holland, 1977.

[Jone78] T. Jones, "Measuring Programming Quality and Productivity", *IBM Systems Journal*, No. 1, 1978, pp. 39-63.

[Knud89] J. Knudsen, and O. Madsen, "Teaching Object-Oriented Programming is More Than Teaching Object-Oriented Programming Languages", *ECOOP '88: Proceedings of the European Conference on Object-Oriented Programming, Lecture Note on Computer Science*, Vol. 322, August 1989, pp. 21-40.

[Kafu87] D. Kafura, and G. Reddy, "The Use of Software Complexity Metrics in Software Maintenance", *IEEE Transactions on Software Engineering*, Vol SE-13, March 1987, pp. 317-324.

[Kors90] T. Korson, and J. McGregor, "Understanding Object-Oriented: A Unifying Paradigm", *Communications of the ACM*, Vol 33 (9), September 1990, pp. 40-60.

[McCa76] T. McCabe, "A Complexity Measure", *IEEE Transactions on Software Engineering*, December 1976, pp. 308-320.

[Morr89] K. Morris, *Metrics for Object-oriented Software Development Environments*, Masters Thesis, M.I.T., 1989.

[Oman90] P. Oman, and C. Cook, "Design and Code Traceability", *The Journal of Systems and Software*, Vol 12 (3), July 1990.

[Perr90] D. Perry, and G. Kaiser, "Adequate Testing in Object-Oriented Programming", *Journal of Object-Oriented Programming*, Vol 2 (5), January/February 1990, pp. 13-19.

[Pohl89] I. Pohl, *C++ for C Programmers*, Benjamin/Cummings, Menlo Park, California, 1989.

[Poun90] D. Pountain, "Object-Oriented Programming", *BYTE*, February 1990, pp. 257-264.

[Shen85] V. Shen, T. Yu, S. Thebaut, and L. Paulsen, "Identifying Error-Prone Software - An Empirical Study", *IEEE Transactions on Software Engineering*, Vol SE-11, April 1985, pp. 317-324.

[Seid86] E. Seidewitz, and M. Stark, "Towards a General Object-Oriented Software Development Methodology", *Proceedings of the First International Conference on Ada Programming Language Applications for the NASA Space Station*, 1986, pp. D.4.6.1-D.4.6.14.

[Taen89] D. Taenzer, M. Ganti, and S. Podar, "Object-Oriented Software Reuse: The Yoyo Problem", *Journal of Object-Oriented Programming*, Vol. 2 (3), September/October 1989, pp. 30-35.

[Wegn87] P. Wegner, "Dimensions of Object-based Language Design", Special Issue of *SIGPLAN Notices*, Vol 22 (12), October 4-8, 1987, pp. 168-182.

[Weyu88] E. Weyuker, "Evaluating Software Complexity Metrics", *IEEE Transactions on Software Engineering*, Vol. 14 (9), September 1988, pp. 1357-1356.

[Wirf90] R. Wirfs-Brock, and R. Johnson, "Surveying Current Research in Object-Oriented Design", *Communications of the ACM*, Vol 33 (9), September 1990, pp. 104-124.

# Appendix A
# Background questionnaire

Instructions: You have 10 minutes to complete this background questionnaire sheet and read the instructions on the next page. Once you have finished the background questionnaire read the instructions on the next sheet. Once you have completed the background questionnaire do not come back to this page during the experiment.

Graduate Major: _____

Undergraduate degree in Computer Science Yes / No (circle one)

Number of years in grad school _____

Number of years programming: _____

Number of years programming professionally: _____

List the programming languages you feel you are proficient in

_____

_____

_____

_____

_____

How experienced a C++ programmer are you? (circle one)

(expert) 10 9 8 7 6 5 4 3 2 1 (novice)

How knowledgeable are you in the object-oriented paradigm? (circle one)

(expert) 10 9 8 7 6 5 4 3 2 1 (novice)

Thanks for your assistance with this experiment.

Go on to the next page when you have completed this page.

## Appendix B Raw Data

| Item | | Group A | | | | | | Group B | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Background information | Subject number | A-1 | A-2 | A-3 | A-4 | A-5 | A-6 | B-1 | B-2 | B-3 | B-4 | B-5 |
| | Grad major | CS | CS | CS | CS | CS | CS | CE | CS | CS | CS | CS |
| | Undergrad CS | No | Yes | No | Yes | Yes | Yes | No | No | No | No | Yes |
| | # yrs grad | 1.5 | 1 | 2.5 | 1 | 1 | 2 | 2 | 3.5 | 1 | 2 | 1 |
| | # yrs prg | 4 | 5 | 9 | 5 | 4 | 6 | 3 | 1.5 | 7 | 15 | 6 |
| | yrs professional | 0 | 1 | 6.5 | 1.5 | 0 | 6 | 0 | 0 | 3 | 8 | 1 |
| | C++ proficient | 8 | 4 | 8 | 6 | 4 | 8 | 5 | 5 | 7 | 5 | 3 |
| | OOP proficient | 10 | 7 | 8 | 8 | 4 | 8 | 6 | 8 | 7 | 8 | 3 |
| Comprehension task | tree version | deep | | | | | | shallow | | | | |
| | time | 20 | 16 | 18 | 9 | 20 | 20 | 20 | 19 | 20 | 12 | 20 |
| | # answered | 4 | 4 | 4 | 4 | 2 | 4 | 3 | 4 | 4 | 4 | 4 |
| | correctness@ | 2 | 3 | 3 | 2 | 2 | 2 | 2 | 2 | 3 | 3 | 2 |
| Program error task | tree version | shallow | | | | | | deep | | | | |
| | time | 10 | 3 | 2.5 | 4 | 3 | 2 | 9 | 10 | 2 | 4 | 10 |
| | identification* | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 0 |
| | correction+ | 2 | 3 | 3 | 3 | 2 | 2 | 2 | 0 | 1 | 3 | 0 |
| Program modification task | tree version | deep | | | | | | shallow | | | | |
| | time | 15 | 15 | 11 | 10 | 10 | 11 | 15 | 15 | 13 | 15 | 15 |
| | identification* | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 1 |
| | change ~ | 2 | 1 | 2 | 3 | 1 | 1 | 3 | 3 | 3 | 3 | 2 |
| Conclusions | Confidence | 2 | 3 | 3 | 4 | 3 | 2 | 2 | 3 | 4 | 3 | 1 |
| | Class list use& | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| | Class list useful& | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

| Legend | * identification | @ correctness | + correction | ~ change | &use |
|---|---|---|---|---|---|
| | 0=Incomplete/Incorrect area | 0=no answer | 0=no answer attempted | 0=no answer attempted | 0=No |
| | 1=Complete | 1=wrong answer | 1=correction attempted in wrong place | 1=change attempted in wrong area | 1=Yes |
| | | 2=numbers correct, wrong order | 2=in right place, error generated | 2=in right place, error generated | |
| | | 3=answer correct | 3=correction attempted, correct | 3=change attempted, correct | |