Object-Oriented Software Quality Through Data Scope Complexity Measurement

Ying-Hong Wang, Chi-Ming Chung, Timothy K. Shih,
Huan-Chao Keh and Wei-Chuan Lin
Graduate Institute of Information Engineering
TamKang University, Tamsui, Taiwan, 25137, R.O.C.
E-Mail: inhon@cs.tku.edu.tw

Abstract

Software metrics is a necessary step for software reliability and quality and software metrics technique of traditional procedure-oriented programming is fairly maturity and has various methodologies and tools available for use. Recently, object-oriented programming became popular. However, traditional procedure-oriented software metrics are not appropriate for the development of an object-oriented software. Some researches of object-oriented software metrics have been proposed. But, these articles focus on only one metric that measures a specific characteristic of the object-oriented software. In this paper, we propose a new metric methodology, the ***data scope complexity****, for object-oriented software based on data scope of a program. The data scope complexity can show complexities of multiple features of object-oriented programming at the same time. Also, we quantify and compare object-oriented programming with procedure-oriented programming.*

Index items: Object-oriented programming (OOP), Software metrics, Data scope, C++ programming language, Friend function, Public variable, Protected variable, Private variable

1. Introduction

Software metrics is a necessary step for software reliability and quality. The general definition of software metrics is to measure the complexity of software. Complexity measurement of program provides a proper norm to evaluate reliability and quality of software.

Since object-oriented programming was widely advocated for the past decade, it has been the major tendency of software development, including object-oriented analysis (OOA) [6], object-oriented design (OOD) [1; 7], and object-oriented programming (OOP) [3; 8], in the 1990's. It contributes to software reusability, software flexibility and software extensibility [9-10; 12-14]. It also increases software reusability, software flexibility and software extensibility.

Although software metrics technique of traditional procedure-oriented programming is fairly maturity and has various methodologies and tools available for use. However, the traditional procedure-oriented software metrics is not appropriate for object-oriented software, due to the different program features of object-oriented programming. Therefore, developing a software metrics for **OOP** is desirable.

Data scope in a programming language is used to define the visibility of variables in a code segment. Due to the feature of encapsulation, the data scope concept of OOP is very different comparing with traditional procedure-oriented programming language. In our paper, we focus the data scope of object-oriented programming language and propose a methodology based on data scope to measure the complexity of a object-oriented program. At the same time, we present the advantage of object-oriented programming to contrast with procedure-oriented language through data scope complexity.

The rest of the paper consists of the following. The related investigations are presented in Section 2. Section 3 describes the data scope mechanism of the C++ programming language. The data scope complexity is proposed in Section 4. Comparing and proof the advantage of object-oriented programming with procedure-oriented programming are given in

0-7803-4053-1/97/$10.00 © 1997 IEEE

Section 5. Section 6 addresses our conclusions and future works.

2. Survey of Related works

For object-oriented software production to fulfill its promise in moving software development and maintenance from the current "craft" environment into something more closely resembling conventional engineering, it requires measures or metrics of the process. Chidamber [4] presents a suit of metrics for object-oriented software. These metrics are based on measurement theory and informed by the insights of experienced object-oriented software developers. Chung [5] also presents a suit of object-oriented metrics based on property of inheritance.

2.1 Chidamber's Measurement Theory

Chidamber proposed five metrics from different views to the characteristics of object-oriented software. They are describes as follows :

Metric 1: Weighted Methods per Class (WMC)

Definition 2-1: Consider a class C, with methods M_1, M_2, ... M_n, and let c_1, c_2, ..., c_n be the static complexity of the methods. Then WMC of C is

$$WMC = \sum_{i=1}^{n} C_i$$

if all static complexities are considered to be unity, WMC = n, which is the number of methods.

Metric 2: Number Of Children (NOC)

Definition 2-2: NOC is the number of immediate sub-classes subordinated to class in a class hierarchy.

Metric 3: Coupling Between Objects (CBO)

Definition 2-3: CBO for a class is a count of the number of non-inheritance related couples with other classes.

Metric 4: Response For a Class (RFC)

Definition 2-4: RFC = $| RS |$, where RS is the response set for a class.

Metric 5: Lack of Cohesion in Method (LCOM)

Definition 2-5: Consider a class C with methods M_1, M_2, ... M_n. Let $\{I_1\}$ be a set of instance variables used by method M_i. There are n such sets $\{I_1\}$, $\{I_2\}$, ... $\{I_n\}$. LCOM is the number of disjoint sets formed by the intersection of n sets.

2.2 Chung's Inheritance-Based Metrics

Chung et al. [5] present a family of inheritance-based metrics to measure class hierarchy complexity. They are shown below.

Metric A: Depth of Inheritance Level (DIL)

Definition 2-6: Consider a class hierarchy $G = (V, E)$, where $| E | = k$, and $| V | = n$. DIL is the longest inheritance path in G [11].

Metric B: Number of Inheritance Edge (NIE)

Definition 2-7: NIE for a class hierarchy is a count of the number of inheritance edges. It is easy to verify that G_1, with more edges, is more complex than G_2, with less edges, even if G_1 and G_2 have the same DIL value.

Metric C: Combination of NIE and DIL (CND)

These two metrics, DIL and NIE, could be used as the basic measurement unit. Given a class hierarchy G, it can be defined that a metric $CND(G)$ as a linear combination of $NIE(G)$ and $DIL(G)$.

$$CND(G) = \alpha \times DIL(G) + \beta \times NIE(G)$$

where α and β are two coefficients. The value of α and β could vary according to different object-oriented systems. If an object-oriented system with a high depth tendency, α could be chosen as a bigger value to emphasize the property. On the contrary, β could be treated in the same manner. Interestingly, given a class hierarchy, $G = (V, E)$, the complexity of G will be located in a closed region formed by $DIL(G)$ and $NIE(G)$. The complexity of class hierarchy could be characterized in this area.

3. The Principles of Data Scope Mechanism in C++

Object-oriented programming essentially means programming using ***objects*** and other concepts. The *object* concept is the most important concept that an object-oriented language must support. The language must support the definition of a set of operations for

the object, namely the object's interface, as an implementation part for the object, which a user of the object should not know about. The object implementation is thus encapsulated and hidden from the user.

There exists many object-oriented programming languages, such as Smalltalk, C++, Eiffel, Object-C, Simula, etc. To simplify our description, we use C++ in our examples. In the C++ programming language, an object properties are implemented internally as a number of variables, called *attributes*, which store information and a number of functions, called *methods*, which access or update the contents of these variables. The variables are divided into three types: ***private***, ***protected*** and ***public***. The *private* variables can be accessed only by member of the object. The *public* variables are accessed by all objects. The inheriting objects can access the public and *protected* variables and functions. The functions of the object itself can access variables of all types. In the following example, class *basescore* is a superclass that includes three types of variables and a public function. The class *dbase* is a subclass that inherits from *basescore* and overrides the public function. The main function defines an object *pub* is a *dbase* class and calls the public function of pub.

Example :

```
#include <iostream.h>   //for cout, cin

class basescore     //base class for score
{
    private :
        int math;
    protected :
        int eng;
    public :
        int chem;
 void input( )
 {
  cout << " Input Mathematic score : "; cin >>
math;
  cout << " Input English score : "; cin >> eng;
  cout << " Input Chemical score : "; cin >> chem;
 }  /* End of method definition, all three data type
can be accessed */
} //End of basescore class definition

class dbase : public basescore   /* class dbase
inherited from basescore */
{
  public :
 void input( )   // Overloading input function
 {
// cout << " Input Mathematics score : "; cin >>
math;
/* It is error because of inherited class just can
access protected and public data */
     cout << " Input English score : "; cin >> eng;
     cout << " Input Chemical score : "; cin >>
chem;
 } /* End of method definition, it can access
protected and public data */
} //End of dbase class definition

main( )
{
     dbase pub;
     int s;
     pub.input( );
//   s = pub.math;
/* It is error because of it just can access public data,
math is a private data */
//   s = pub.eng;
/* It is error because of it just can access public data,
eng is a protected data */
     s = pub.chem;
/* It is correct because of it can access public data,
chem is a public data */
} // End of main function
```

When the program is executed, only *input* function declared in *dbase* class is active. Thus the user only input the score of English and Chemical. The reason is that *pub* is an object of *dbase* class and the *dbase* class is inherited from the *basescore* class. Therefore, *dbase* class can only access protected and public variables, namely *eng* and *chem*. The variable, *math*, is *basescore's* private variable. The input function of *dbase* class overrides from the *basescore* class and redeclares to access *eng* and *chem* variables only. The *main* function is a independent function. Therefore, it only call public function and public variable.

4. Object-Oriented Software Quality based on Data Scope

In this section, a perspective of software quality from data scope will be presented. Then, the data scope complexity of object-oriented programs and procedure-oriented programs are also compared. To simplified our descriptions, we use C++ and C as the instances of object-oriented programming language and procedure-oriented programming language.

In the C++ programming language, each object consists of variables (i.e. attributes) and functions (i.e. methods). Each object has three types of variables: *private*, *protected* and *public*. And each object can have some ***friend*** functions. The friend functions are not member functions of the object but able to access any types of variables. Let each object consists of the following elements:

V_{pub} : the number of public variables in the object.
V_{prt} : the number of protected variables in the object.
V_{pri} : the number of private variables in the object.
F : the number of functions in the object.
O_{sub} : the number of objects inherited from the object, including direct and indirect inheritance.
F_{fri} : the number of *friend* functions in the object.

From the perspective of data scope, if an object-oriented program contains N objects and in the worst case that every object has relationship with the other all objects. When the object-oriented program is executed, the situation of variables accessed in an object will be

$V_{pri} * F + (\Sigma^{Osub}_{i=1} F_i + F) * V_{prt} + \Sigma^{N}_{j=1} F_j * V_{pub} + (V_{pri} + V_{prt} + V_{pub}) * F_{fri}$.

We call the equation the *data scope complexity* of an object, D_{obj}. The first term, $V_{pri} * F$, presents the *data scope* of private variables are only active in all functions of self-object. The second term, $(\Sigma^{Osub}_{i=1} F_i + F) * V_{prt}$, means the *data scope* of protected variables are not only active in all functions of self-object but also active in all functions of sub-object inherited from this object. The third term, $\Sigma^{N}_{j=1} F_j * V_{pub}$, means the *data scope* of public variables are active in all functions of all objects, and the last term, $(V_{pri} + V_{prt} + V_{pub}) * F_{fri}$, presents the *data scope* of all type variables of an object are active in its friend functions. Since the object-oriented program has N objects, the *data scope complexity*, D_{pgm}, of the program is the summation of D_{obj} of all objects, shown as follows:

$D_{pgm} = \Sigma^{N}_{k=1} D_{objk}$, that is

$\Sigma^{N}_{k=1}(V_{prik} * F_k + (\Sigma^{Osub}_{i=1} F_i + F_k) * V_{prtk} + \Sigma^{N}_{j=1} F_j * V_{pubk} + (V_{prik} + V_{prtk} + V_{pubk}) * F_{frik})$

where *k*, from 1 to N, means the data scope complexity of *k*-th object, simultaneously it means all private, protected, and public variables, member functions and friend functions of *k*-th object. The F_i of $\Sigma^{Osub}_{i=1} F_i * V_{prtk}$ means that the number of functions of the *i*-th object which inherits from the *k*-th object.

The *data scope complexity* D_{pgm} reacts to some facts that described below:

1. When an object adds a friend function, the D_{pgm} will be increased by a value which is the number of variables declared in the object. It represents that the friend functions will destroy the *encapsulation* feature of object-oriented programming and increase the complexity of a program, even if friend functions provide programmer to flexible programming. The part of data scope complexity, $(V_{pri}+V_{prt}+V_{pub}) * F_{fri}$, presents the specially feature of the C++ programming.
2. Public variables are another factor that affects the D_{pgm}. It represents an object shares its variables with other objects. It is called the ***coupling*** between objects. The part of data scope complexity, $\Sigma^{N}_{j=1} F_j * V_{pub}$, is similar to CBO and RFC metrics proposed by Chidamber [4].
3. **Inheritance** is a major characteristic of object-oriented programming. Generally it is better to have the number of inherited objects than the number of equalized objects, since it promotes reusability, flexibility and extensibility of objects through inheritance. However, the more depth or breadth inherited objects mean the more testing effort will be spent, since of the error in the super-object could be propagated through inheritance. The part of data scope complexity, $\Sigma^{Osub}_{i=1} F_i * V_{prt}$, is similar to NOC metric proposed by Chidamber [4] and CND metric proposed by Chung [5].

5. Comparing with Procedure-Oriented Programming

In general, one program is implemented in the C++ programming language. It can be also implemented in the C programming language. **Hypothesis:** there are existed a C++ program and a C program implemented for same task. The C++ program is divided into *N **objects*** and the C program is divided into *N **modules***. Let C program has same total number of variables and functions as the C++ program. However, the C programming language has no concept of object. The data scope of C is divided into **local** variables and **global** variables. Therefore, the protected and public variables in the C++ program will be accessed by different functions in the C program, i.e., the protected and public variables will become to the global variables in the C program. In the best case, each private variable may be accessed by one function in a C++ program, they

can be distributed and become the local variables of some individual functions in a C program. In the worst case, each private variable is accessed by two or more functions in the C program, then the private variables are still became global variables. Another speaking, in the best case, the private variables of C++ program will become local variables in the C program. But in the worst case, they are became global variables. Supposing there is no friend functions in the C++ program. Because if there exists any friend function, it means that the private variables would be shared by two or more functions and the private variables would become global variables in the C program. When the procedure program coded by C is executed, the situation of variables accessed among functions will be

In the best case:

$$D_{pgm} = \Sigma^{N}_{k=1}(V_{prik} * 1 + \Sigma^{N}_{j=1}F_j * (V_{prtk} + V_{pubk})),$$

In the worst case:

$$D_{pgm} = \Sigma^{N}_{k=1}(\Sigma^{N}_{j=1}F_j * (V_{prik} + V_{prtk} + V_{pubk})),$$

where k, from 1 to N, means k-th module of the C program. The $V_{prik} * 1$ means the number of private variables V_{prik} can only be accessed by one function in the C program.

Comparing the data scope complexity D_{pgm} of C++ programs with the one of C programs, we present a theorem according to the quantitative data scope equation and prove the advantage of object-oriented programming over procedure-oriented programming.

Theorem :

The *data scope complexity* of object-oriented programming is less then procedure-oriented programming.

proof :

The *data scope complexity* of C++ program is shown as Equation (1).

(1). $D_{pgm} = \Sigma^{N}_{k=1}(V_{prik}*F_k + (\Sigma^{Osub}_{i=1}F_i+F_k)*V_{prtk} + \Sigma^{N}_{j=1}F_j*V_{pubk})$

The *data scope complexity* of C program is shown as Equations (2) and (3):

In the best case:

(2). $D_{pgm} = \Sigma^{N}_{k=1}(V_{prik} * 1 + \Sigma^{N}_{j=1}F_j * (V_{prtk} + V_{pubk}))$

In the worst case:

(3). $D_{pgm} = \Sigma^{N}_{k=1}(\Sigma^{N}_{j=1}F_j * (V_{prik} + V_{prtk} + V_{pubk}))$

Note: The data scope complexity of C++ program has no $(V_{pri}+V_{prt}+V_{pub})*F_{fri}$ because that we suppose there does not exist friend functions in the C++ program.

Comparing the ***data scope complexities*** of C++ program(Object-oriented programming) with C program (Procedure-oriented programming), we can use D_{pgm} of C++ program to subtract D_{pgm} of C program.

In the best case, (1) - (2) :

(4). $\Sigma^{N}_{k=1}(V_{prik} * (F_k - 1) + (\Sigma^{Osub}_{i=1}F_i + F_k - \Sigma^{N}_{j=1}F_j)* V_{prtk}) = \Sigma^{N}_{k=1}(V_{prik} * (F_k - 1) + (\Sigma^{Osub}_{i=1}F_i - \Sigma^{N}_{j=1, j*k}F_j)* V_{prtk})$

In the worst case, (1) - (3) :

(5). $\Sigma^{N}_{k=1}((F_k - \Sigma^{N}_{j=1}F_j)* V_{prik} + (\Sigma^{Osub}_{i=1}F_i + F_k - \Sigma^{N}_{j=1}F_j)*V_{prtk}) = \Sigma^{N}_{k=1}((F_k - \Sigma^{N}_{j=1}F_j)* V_{prik} + (\Sigma^{Osub}_{i=1}F_i - \Sigma^{N}_{j=1, j*k}F_j)*V_{prtk})$

Recall the symbols F_k represents the number of functions in *k*-th object, O_{sub} is the number of objects inherited from *k*-th object, N means the number of objects. Due to the O_{sub} only is the number of inherited objects that belongs to a specific object, the N is the number of all objects in a program. In normal situation of programming, the O_{sub} should less than N. Same situation, the F_k only is the number of functions of a specific object and the $\Sigma^{N}_{j=1, j*k}F_j$ is the number of total functions of all objects, so the F_k should much less than $\Sigma^{N}_{j=1, j*k}F_j$. Therefore,

$O_{sub} < N$, $\Sigma^{Osub}_{i=1}F_i << \Sigma^{N}_{j=1, j*k}F_j$, and $F_k << \Sigma^{N}_{j=1, j*k}F_j$.

$=>$ In Equation (4),

$\Sigma^{Osub}_{i=1}F_i - \Sigma^{N}_{j=1, j*k}F_j << 0$, and $V_{prik} * (F_k - 1) + (\Sigma^{Osub}_{i=1}F_i - \Sigma^{N}_{j=1, j*k}F_j)* V_{prtk} < 0$.

$=>$ In Equation (5),

$F_k - \Sigma^{N}_{j=1}F_j << 0$, $\Sigma^{Osub}_{i=1}F_i - \Sigma^{N}_{j=1, j*k}F_j << 0$, and $(F_k - \Sigma^{N}_{j=1}F_j)* V_{prik} + (\Sigma^{Osub}_{i=1}F_i - \Sigma^{N}_{j=1, j*k}F_j)*V_{prtk} << 0$.

So the theorem "the *data scope complexity* of object-oriented programming is less then procedure-oriented programming" is proved. ■

One thing needs to mention, the ***data scope complexity*** of object-oriented programming is in the worst **hypothesis:** every object has relationship with the other all objects. It means the data scope complexity, D_{pgm}, is the *maximum* value of an object-oriented program.

6. Conclusions and Future Work

Recently, object-oriented programming gradually became popular. However, traditional procedure-oriented software metrics are not appropriate for the development of an object-oriented software. In this paper, we present a metric methodology for object-oriented programs through data scope viewpoint. Also, the data scope complexity responses simultaneously some of Chidamber's measurement theory and Chung's inheritance-based metrics. On the other hand, we use the data scope complexity to quantify the advantage of object-oriented programming comparing to procedure-oriented programming.

There are two directions in our future works. One is extending the metrics research to testing methodologies for object-oriented software. The other is to discuss how to integrate these object-oriented software metrics and testing methodologies to be a software development tools.

Reference

1. [Booch 86] Grady Booch, "Object-Oriented Development," IEEE Trans. Software Eng., Vol. 2, No. 2, Feb. 1986

2. [Booch 94] Grady Booch' "Object-Oriented Analysis and Design with Applications," 2nd Edition, Benjamin/Cummings Publishing CO., INC.

3. [Budd 91] Timothy Budd, "Object-oriented Programming," Addison-Wesley, 1991

4. [Chidamber 91] Shyam R. Chidamber and Chris F. Kemerer, "Toward Metrics Suit for Object Oriented Design," OOPLSA'91, pp.197-211

5. [Chung 94] Chi-Ming Chung and Ming-Chi Lee, "Object-Oriented Design Complexity Metrics," International Journal of Mini and Microcomputer, Vol. 16, No. 1, pp. 7-15, Jan., 1994

6. [Coad 90] Peter Coad and Edward Yorudon, "Object-Oriented Analysis," Prentice Hall, 1990

7. [Coad 91] Peter Coad and Edward Yorudon, "Object-Oriented Design," Prentice Hall, 1991

8. [Cox 87] Brad J. Cox, "Object-Oriented Programming," Addison-Wesley, 1987

9. [Jacobson 92] Ivar Jacobson, Magnus Christerson, Patrik Jonsson, Gunnar Overgaard, "Object-Oriented Software Engineering," Addison-Wesley, 1992

10. [Laranjeira 90] L. A. Laranjeira, "Software Size Estimation of Object-Oriented Systems," IEEE Trans. on Software Eng., Vol. 16, No. 5, pp. 510-522, May, 1990

11. [Moreau 89] D. R. Moreau, and W. D. Dominick, "Object-Oriented Graphical Information System: Research Plan and Evaluation Metrics," Journal System and Software, Oct. 1989, pp.23-28

12. [Rumbaugh 91] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy and William Lorensen, "Object-Oriented Modeling and Design," Prentice Hall, 1991

13. [Seidewitz 89] E. Seidewitz, "General object-oriented software development: background and experience," Journal System and Software, No. 19, pp. 95-118, 1989

14. [Ward 89] P. T. Ward, "How to Integrate Object-Orientation with Structured Analysis and Design," IEEE Software, No. 6, pp. 74-82, March 1989

15. [Wilkie 93] George Wilkie, "Object-Oriented Software Engineering -- The Professional Developer's Guide," Addison-Wesley, 1993.