IEEE Region 10 Conference, Tencon 92
11th - 13th November, 1992
Melbourne, Australia

Inheritance-Based Object-oriented Software Metrics

Chi-Ming Chung and Ming-Chi Lee

Department of Computer Science
Tamkang University
Taipei, Taiwan, R.O.C.

Abstract

Object-oriented software development, including *object-oriented analysis* (OOA), *object-oriented design* (OOD) and *object-oriented programming* (OOP), is a new promising approach for developing software systems to reduce software costs and to increase software reusability, flexibility, and extensibility. *Software metric* is an important technique used to measure software complexity to improve software quality and enhance software correctness. However, there is still no software metrics based on object-oriented programming languages (**OOPLs**) developed to help object-oriented software development. This paper describes a graph-theoretical complexity metric to measure object-oriented software complexity. It shows that *inheritance* has a close relation with the object-oriented software complexity, and reveals that misuse of *repeated (multiple)* inheritance will increase software complexity and be prone to implicit software errors. An algorithm to support this software metric is presented. Its time complexity is $O(n^3)$.

1 Introduction

Object-oriented software development [Booch 86][Coad 90] is a new promising approach for developing software systems to reduce software cost and to improve software productivity. It has been the major tendancy of software development in 1990's but it is still lacking of testing methodologies and software metrics based on the OOPLs. In the software lifecycle, there are two important techniques to insure software reliability and quality. One is through software testing to minimize errors, and the other is to utilize software metrics to monitor the software complexity for improving the quality of the program. A high quality software should have the characteristics of understandability and measurability [Adrion 82]. Since complexity is a significant and determinant factor of a system's success or failure, the risk is high for software development by ignoring the complexity measurement. Software metric theory has received growing attention over the past decade. Much of the researches on software metrics has been involved with procedure-oriented languages such as PL/I, Pascal, Fortran and C. Many studies have dealt with the subject of measuring these procedure-oriented program complexity, but few studies have concentrated on measuring the complexity of the *object-oriented software*. Most existing procedure-oriented software metrics are developed from program factors such as program size, control flow and data dependency. However, the program factors of OOPLs are not merely limited on these factors. Generally speaking, an object-oriented programming language must exhibit four program factors (features): *inheritance, data abstraction, dynamic binding, and information hiding* [Pinson 88]. Most of these features do not exist in procedure-oriented programming languages, especially the inheritance does not exist in them. It is desirable to take the features of OOPLs into consideration for the development of object-oriented software metrics. This paper concentrates on the inheritance feature.

Inheritance is the most important feature for software reuse and it supports the *class hierarchy design* and captures the the *is-a relationship* between a class and its subclass. This class hierarchy design, represented by an inheritance graph, has been widely applied to object-oriented software, object-oriented database, and graphics system design [Seidewitz 89][Horowize 91]. Although *repeated inheritance* and *multiple inheritance* allows a class to inherit from more than one parent class to increase reusability, the possibility of confliction between parent classes not only increases software complexity but also leads to implicit software errors [Meyer 88a]. This type errors are called *name-confliction* . In this paper, we present an *inheritance-based* metric to measure the object-oriented software complexity for detecting software errors and improving software correctness. Three graph theorems are proposed and prove that any repeated inheritance graph must consist of a set of *unit repeated inheritances* (URIs) which are the basis of this metric. The more URIs an inheritance graph contains, the more complex and error-prone it would be. This relationship between the inheritance-based metric and OOP software complexity is shown in figure 1.

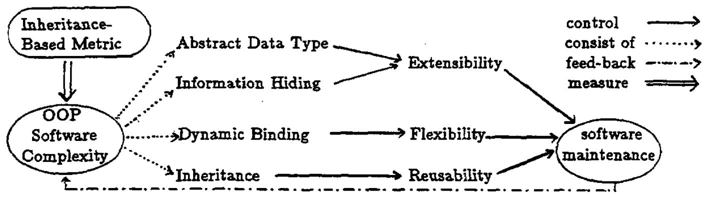


Figure 1. Inheritance-Based Metric and OOP Software Complexity Relation

In Section 2, the procedure-oriented software metrics are briefly introduced. In section 3, the graph theory is applied to describe the inheritance mechanism and three graph theorems to support the inheritance-based metric are proposed. In Section 4, an algorithm to support this metric is presented. In section 5, we propose this metric and illustrate the application of it to measure the class hierarchy design complexity. The future research is recommended in the final section.

2 Software Metrics

Software metrics have been widely used and applied in the field of software development for a number of years. Use of software metrics is well recognized as an effective technique to improve the quality of software. Most metrics are derived from complexity factors such as factors of size, control flow, and data dependency. Therefore, metrics could be classified into size metrics, control flow metrics, and data data flow metrics [Chung 88].

A. Program Size Based Metric

Halstead proposed a family of metrics called software science. It is one of most well known metrics which has been widely accepted and applied. In software science, a program is treated as a group of tokens. Tokens are divided into two classes, one is operands, the other is operators. A family of metrics is derived from the basic counts: *Vocabulary, Length, Length Estimate, Volume, Program Level, Effort, and Language Level* [Halstead 77].

B. Control Flow Based Metric

Most control flow complexity metrics are derived from the control flow graph of a program. The control flow graph is the basis of control flow metrics. A *flow graph PG = (V,E,s,t)* defined as a binary directed graph, where V is a set of vertices reachable from the start vertex, E is a set of directed edges representing the control flows, $s \in V$ is the start vertex with no edge entering it, and $t \in V$ is the terminal vertex with no edge going out.

C. Data Flow Metric

Data flow based metrics are concerned about the inter and intra module's data dependency complexity. Numerous studies show that the data dependency of a program has a significant effect on the programer's tracing, debugging, and understanding capability [Donsmore 79].

3 Inheritance and Graph Theorems

Inheritance is the major mechanism of OOPLs for software reuse which is different from the module reuse, such as subroutine calls or package in Ada [Meyer 88a]. It allows the same code inherited from parent class without any function (subroutine) calls. It also supprots the *class hierarchy* design which captures the *is-a* relationship between a class and its subclass. The class hierarchy is usually represented by a directed graph, called *inheritance graph*.

An inheritance graph could be divided into three basic structures: *1. single inheritance, 2. multiple inheritance,* and *3. repeated inheritance*; they are represented by a connected directed graph G=(V,E), where V is a set of classes, and E is a set of inheritance edges which are ordered relations such that $E = \{ x \rightarrow y \mid y$ inherits from x, where x and $y \in V \}$. Also, there are three types of inheritance edges *tree edges, forward edges,* and *backeges*: tree edges connect parents to children in the graph, and forward edges connect ancestors to decendants. However, *back edges* which connect decendants to ancestors should be avoided. The reason is that using back edge is prone to enter a *cyclic inheritance* [Horowize 91]. Now, we discuss the three basic structures of inheritance graph and present three theorems to help develop the OOP software metrics. First, a single inheritance is that each class inherits uniquely from one parent class. Second, if a class is permitted to inherit from more than one parent class, it is called a *multiple inheritance*[Geib 90].

Lemma 1 : Suppose inheritance graph $G_{mul} = (V, E)$ contains multiple inheritance, where V is a set of classes and E is a set of inheritances edges. Then, there is at least one vertex $v \in V$ whose in-degree is ≥ 2.

For example, If a class A inherits from two parent classes, class B and class C, then the in-edges of class A are 2. However, this case would lead to function name clashes between the inherited classes. If, for example, both B and C contain a function *print*, it is an ambiguity for class A, because A cannot distinguish it [Meyer 88b]. This function clash is called *name-confliction*.

Third, for given a multiple inheritance $G_{mul} = (V, E)$, if there exists a common ancestor class such that the parent classes of V inherit from it, the repeated inheritance is defined as $G_{mul} \cup$ the common ancestor $\cup$ the inheritances edges between the parent classes and the common ancestor.

Lemma 2 : Given a repeated inheritance graph $G_{rep} = (S, F)$, there must exist a multiple inheritance graph $G_{mul} = (V, E)$ which is a subgraph of $G_{rep} = (S, F)$ such that $V \subseteq S$ and $E \subseteq F$ is hold.

Theorem 1: Let $G = (V, E)$ be a repeated inheritance graph, then the vertex numbers of $V \geq 3$ is hold and G contains closed regions.

Lemma 3: If $G = (V, E)$ is an inheritance graph containing repeated inheritance, then the *euler's region number* of $G \geq 2$ is hold (i.e, G contains at least one closed region).

Theorem 2: Let G = (V,E) is an inheritance graph. If it contains repeated inheritances, then the graph G could be decomposed into a set of unit repeated inheritanes (URIs).

4 Algorithm for Finding URIs

In this section, the algorithm for finding unit repeated inheritances (URIs) is proposed. The data structure of inheritance relations are represented by a directed graph G=(V,E), where V is the set of all classes and E is the set of all inheritance edges. To illustrate this algorithm, many definitons are needed.

Definition:
1. root class: it is a class node with no in-edges.
2. terminal class: it is a class node with no out-edges.
3. Ancestor(v): it is a set which records all the ancestor class numbers of v and itself, where v is a class of V.

Algorithm: Finding Unit Repeated Inheritances (URIs)

Input: A set of classes and inheritance relations
Output: Unit repeated inheritances

```
Step 1. Build a directed graph consists of classes and inheritance edges and initialize Ancestor($v$) = $\{v\}$ .

Step 2. Using bread-first traverse for all root
          classes; in the process of traverse, parent ancestor set is
          added into its children ancestor sets.

Step 3. For all terminal classes do
            if the number of the ancestor set $\geq 2$ then
            begin
                Union ($set_i$, $set_j$) { $2 \leq$ i,j $\leq$ n and $set_i \neq set_j$ }
                if common parent is found, record the union set
                    A unit repeated inheritance is found.
                else if the number of ancestor set = 1
                    discard the union set (no URI exists)
            end
            else
                no repeated inheritances exists
            endif
```

Let n be the numbers of total classes, t be the numbers of terminal classes and $1 \leq t \leq n$, *Ancestor(i)* be the ancestor sets of the *i*th class, and s_i be the number of the elements in *Ancestor(i)*, for all class $i \in$ terminal classes and $1 \leq s_i \leq n$. For each terminal class, we can union the ancestor set to find out all the URIs and the time complexity of this algorithm is shown as follows:

$$\begin{aligned}\sum_{i=1}^{t} C\binom{s_i}{2} &= C\binom{s_1}{2} + C\binom{s_2}{2} + \cdots + C\binom{s_t}{2}\\ &\leq C\binom{n}{2} + C\binom{n}{2} + \cdots + C\binom{n}{2}\\ &\equiv O(n^3)\end{aligned}$$

5 Inheritance-Based Metrics

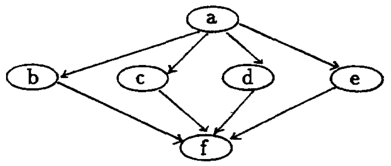
In the previous section, we propose an URI algorithm to help to find out the unit repeated inheritances. In this section, we utilize this algorithm and the graph theorems proposed in section 2 to develop the inheritance-based metric. By theorem 2, we have shown that an inheritance graph is composed of a set of URIs. The more URIs an inheritance graph contains, the more complex and error-prone it will be. Therefore, the complexity of an inheritance graph is defined to be the numbers of the unit repeated inheritances and it is defined as follows:

$$URI(G) = \sum_{i=1}^{t} Union\binom{s_i}{2}(Ancestor(i))$$

where the *Union* operation and *Ancestor* set are defined in section 4.

Two illustrations of this metric are shown in figure 2 and figure 3. In the figure 2, there is one terminal class (t =*1*) and the element numbers of its ancestor set s_1 are 4, so the URI complexity is

$$\sum_{i=1}^{1} Union\binom{s_i}{2}(Ancestor(i)) = 6$$

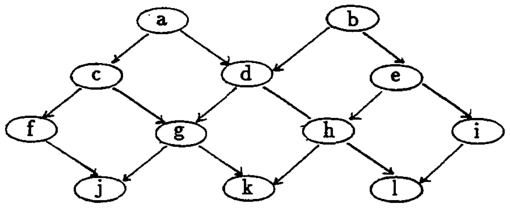


There are six URIs, (abfc), (abfd), (abfe), (acfd), (acfe), and (adfe).

Figure 2. An illustration of inheritance-based metric

In figure 3, there are three terminal classes ($t = 3$) and the element numbers of their ancestor sets are $s_1 = 4$, s_2=6 and s_3=4 respectively. The URI complexity is

$$\sum_{i=1}^{3} Union\binom{s_i}{2}(Ancestor(i)) = 9$$



There are nine URIs, (acgd), (cfjg), (acfjgd), (dgkh), (acgkhd), (bdhe), (bdgkhe), (bdhlie), and (ehli).

Figure 3. An illustration of inheritance-based metric.

6 Conclusion

In this paper, an inheritance-based metric is proposed to measure the complexity of object-oriented software. Also, an algorithm to support this metric is presented. This metric could reflect object-oriented software complexity efficiently to enhance software quality. Furthermore, the proposed algorithm could be implemented to become an useful tool in object-oriented design phase to detect improper inheritance structures.
Further studies based upon this research are : 1. to develope an object-oriented software testing methodology based on this metric to detect the object-oriented software errors. 2. to integrate object-testing to design a new testing tool to build an object-oriented software development environment.

References

[Adrion 82] Adrion, W.R. et al. Validation, verification, and testing computer software, *ACM Comp surv,* 14, 2 (June 1982) pp.160.

[Booch 86] Grady Booch, "Object-Oriented Development," *IEEE Trans. Software Eng.,* vol.2 no.2 ,Feb. 1986.

[Chung 88] Chung, C.M., *A Software Metrics Based Testing Enviroment,*Ph.D. Dissertation, The advanced Center for Computer Studies, Univ, of Southwester Louisisana, Lafayette, Louisiana, 1988.

[Coad 90] Coad, P. and Yourdon, E. *Object-oriented Analysis.* Yourdon Press, 1990.

[Cox 86] Cox, B. *Object-Oriented Programming: An Evolutionary Approach.* Addison-Wesley, New York, 1986.

[Gannon 81] Gannon, J. "Data Abstraction Implementation Specification, and Testing," *ACM Trans on Programming Language System,* vol.3, July, 1981, pp 211-223.

[Geib 90] Geib, Jean-Marc and Carre, Bernard "The Point of View nortion for Multiple Inheritance," *OOPSLA*, 1990.

[Horowize 91] Ellis Horowize and Rajiv Gupta, *Object-Oriented Database with Applications to Case, Networks, and VLSI CAD*, Prenteice Hall 1991.

[Jackson 83] M. Jackson, *System Development.* Englewood Cliffs, NJ:Prentice Hall, 1983.

[Meyer 88a] Betrand Meyer, *Object-Oriented Software Construction,* Prentice Hall 1988.

[Meyer 88b] Betrand Meyer, "Eiffel: A Language and Environment for Software Engineering," *The Journal of Systems and Software 1988.*

[Pascoe 86] Geoffrey A. Pascoe "Elements of Object-Oriented Programming," *BYTE August,1986,* pages 139-144.

[Pinson 88] Lewis J. Pinson. Richard S. Wiener, *An Introduction to Object-oriented Prgramming and Smalltalk,* Addison-Wesley pp 49-60, 1988.

[Seidewitz 89] Seidewitz, E.*General object-oriented software development: background and experience,* J. Syst. and Software. 19, (1989), 95-108.

[Seidewitz 87] Seidewitz, E. and Stark, M." Towards a general object-oriented software development methodology." *Ada Letters,* 7 (July/August 1987) pp 54-67.

[Ward 89] Ward, P. How to integrate object orientation with structured analysis and design. *IEEE Softw.March, 1989), 74-82.*