
Research

Metrics for maintainability of class inheritance hierarchies



Frederick T. Sheldon^{*,†,‡}, Kshamta Jerath and Hong Chung[§]

*Software Engineering for Dependable Systems Laboratory, School of EECS,
Washington State University, Pullman, WA 99164-2752, U.S.A.*

SUMMARY

Since the proposal for the six object-oriented metrics by Chidamber and Kemerer (1994), several studies have been conducted to validate their metrics and have discovered some deficiencies. Consequently, many new metrics for object-oriented systems have been proposed. Among the various measurements of object-oriented characteristics, we focus on the metrics of class inheritance hierarchies in design and maintenance. As such, we propose two simple and heuristic metrics for the class inheritance hierarchy for the maintenance of object-oriented software.

In this paper we investigate the work of Chidamber and Kemerer (1994) and Li (1998), and extend their work to apply specifically to the maintenance of a class inheritance hierarchy. In doing so, we suggest new metrics for understandability and modifiability of a class inheritance hierarchy. The main contribution here includes the various comparisons that we have made. We discuss the advantages over Chidamber and Kemerer's (1994) metrics and Henderson-Sellers's (1996) metrics in the context of maintaining class inheritance hierarchies. Copyright © 2002 John Wiley & Sons, Ltd.

KEY WORDS: software metrics; object-oriented; class inheritance; software maintenance

1. INTRODUCTION

Measurement is fundamental to any engineering discipline and software engineering is no exception [1]. Typically, metrics are essential to software engineering for measuring software

*Correspondence to: Dr Frederick T. Sheldon, School of EECS, Washington State University, Pullman, WA 99164-2752, U.S.A.

†E-mail: sheldon@acm.org

‡F. T. Sheldon is currently on leave at DaimlerChrysler Research and Technology in System Safety, Stuttgart.

§Hong Chung is a visiting scholar from the School of Computer and Electronics, Keimyung University, Taegu, 704-701, Korea.

Contract/grant sponsor: Keimyung University

Contract/grant sponsor: Intel

Contract/grant sponsor: Microsoft

Contract/grant sponsor: DaimlerChrysler



complexity and quality, estimating cost and project effort to simply name a few. The traditional metrics like function point, software science and cyclomatic complexity have been well used in the procedural paradigm. However, they do not readily apply to aspects of the object-oriented (OO) paradigm: class, inheritance, polymorphism, etc.

For about one decade, researchers have been discussing whether a separate set of OO software metrics is needed and what this set should include [2]. Initial proposals focused more towards extension of existing software metrics for procedure-oriented programming [3,4]. However, almost all the recent proposals have focused on OO programming [5–8].

Since the proposal of the six OO metrics by Chidamber and Kemerer (CK) [7] in 1991, other researchers have made efforts to validate the metrics both theoretically and empirically. CK's revised paper [8] proposed a suite of OO metrics which have a set of six simple measures:

- (1) weighted methods per class (WMC) which counts the number of methods in a class;
- (2) depth of inheritance tree (DIT) which is the number of ancestor classes that can affect a class;
- (3) number of children (NOC) which is the number of subclasses that inherit the methods of a parent class;
- (4) coupling between object (CBO) classes which is a count of the number of other classes to which it is coupled;
- (5) response for a class (RFC) which is a set of methods that can be executed in response to a message received by an object of that class; and
- (6) lack of cohesion in methods (LCOM) which is a count of the inter-relatedness between portions of a program.

These metrics were evaluated analytically against Weyuker's measurement theory principles [9] and an empirical sample of these metrics was provided from two commercial systems.

Several studies have been conducted to validate CK's metrics. Basili *et al.* [10] presented the results of an empirical validation of CK's metrics. Their results suggest that five of the six of CK's metrics are useful quality indicators for predicting fault-prone classes. Tang *et al.* [11] validated CK's metrics using real-time systems and the results suggest that WMC can be a good indicator for faulty classes and RFC is a good indicator for faults. Li [12] theoretically validated CK's metrics using a metric-evaluation framework proposed by Kitchenham *et al.* [13]. He discovered some of the deficiencies of CK's metrics in the evaluation process and proposed a new suite of OO metrics that overcome these deficiencies. Balasubramanian [1] identified some deficiencies with the approach taken by CK and proposed some new metrics. Briand *et al.* [6] proposed a new suite of coupling measures for OO design that was empirically validated using a logistic regression technique. Briand and Morasoa have also suggested that these OO coupling measurement metrics are complementary quality indicators to CK's metrics.

Among the various measurements, we focus on the metrics of class inheritance hierarchies in design and maintenance. Class design is central to the development of OO systems. Because class design deals with the functional requirements of the system, it is the highest priority in Object Oriented Design (OOD). Inheritance is a key feature of the OO paradigm. The use of inheritance is claimed to reduce the amount of software maintenance necessary and ease the burden of testing [8] and the reuse of software through inheritance is claimed to produce more maintainable, understandable and reliable software [10]. However, industrial adoption of academic metrics research has been slow due to, for example, a lack of a perceived need. The results of such research are not typically applied



to industrial software [14], which makes validation a daunting and arduous task. For example, the experimental research of Harrison *et al.* [15] indicates that a system not using inheritance is better for understandability or maintainability than a system with inheritance. However, Daly's experiment [16] indicates that a system with three levels of inheritance is easier to modify than a system with no inheritance.

Ordinarily, it is agreed that the deeper the inheritance hierarchy, the better the reusability of classes, but the higher the coupling between inherited classes, the harder it is to maintain the system. The designers may tend to keep the inheritance hierarchies shallow, forsaking reusability through inheritance for simplicity of understanding [8]. So, it is necessary to measure the complexity of the inheritance hierarchy to conciliate between the depth and shallowness of it. We propose two simple and heuristic metrics for the class inheritance hierarchy in the maintenance of OO software.

The rest of this paper is organized as follows. Section 2 presents some related research, especially the work by CK and Li. We propose new metrics for maintenance of class inheritance hierarchies associated with understanding and modifying OO software systems in Section 3 and discuss them in Section 4. A summary and concluding remarks are presented in Section 5, while Section 6 includes future plans regarding validation.

2. RELATED WORK

Almost all researchers note the need to measure inheritance structure in terms of depths and class density. This can be measured as the depth of each class within its hierarchy, since this is likely to affect the distribution of inherited features. Briand *et al.* [17] empirically discovered that the depth of a class in its inheritance hierarchy appears to be an important quality factor.

CK [8] proposed the DIT metric, which is the length of the longest path from a class to the root in the inheritance hierarchy and the NOC metric, which is the number of classes that directly inherit from a given class. Henderson-Sellers [18] suggested the AID (average inheritance depth) metric, which is the mean depth of inheritance tree and is an extension of CK's DIT. Li [12] suggested the NAC (number of ancestor classes) metric to measure how many classes may potentially affect the design of the class because of inheritance and NDC (number of descendent classes) metric to measure how many descendent classes the class may affect because of inheritance. Tegarden *et al.* [3] proposed the CLD (class-to-leaf depth) metric, which is the maximum number of levels in the hierarchy that are below the class and the NOA (number of ancestor) metric, which is the number of classes that a given class directly or indirectly inherits from. Lake and Cook [19] suggested the NOP (number of parents) metric, which is the number of classes that a given class directly inherits from and the NOD (number of descendents) metric, which is the number of classes that directly or indirectly inherit from a class.

Among the metrics related to class inheritance hierarchy, we discuss in detail the work of CK [8] and Li [12].

2.1. CK's DIT and NOC definitions

Consider the definition for the DIT metric.

- *Definition.* The depth of inheritance of a class is the DIT metric for the class. In cases involving multiple inheritance, the DIT will be the maximum length from the node to the root of the tree.

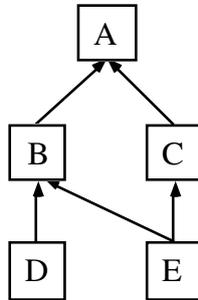


Figure 1. A class inheritance tree.

- *Theoretical basis.* The DIT metric is a measure of how many ancestor classes can potentially affect this class.
- *Viewpoints.*
 - (1) The deeper a class is in the hierarchy, the higher the degree of methods inheritance, making it more complex to predict its behavior.
 - (2) Deeper trees constitute greater design complexity, since more methods and classes are involved.
 - (3) The deeper a particular class is in the hierarchy, the greater the potential reuse of inherited methods.
- *Example.* Consider the class inheritance tree in Figure 1, $DIT(A) = 0$ because A is the root class. $DIT(B) = DIT(C) = 1$ because the length from class B and C to the root A is one each. And, $DIT(D) = DIT(E) = 2$ because the maximum length from class D and E to the root A are two each.

Consider the definition for the NOC metric.

- *Definition.* NOC is the number of immediate subclasses subordinate to a class in the class hierarchy.
- *Theoretical basis.* NOC is a measure of how many subclasses are going to inherit the methods of the parent class.
- *Viewpoints.*
 - (1) The greater the number of children, the greater the potential for reuse, since inheritance is a form of reuse.
 - (2) The greater the number of children, the greater the likelihood of improper abstraction of the parent class.
 - (3) The number of children gives an idea of the potential influence a class has on the design.

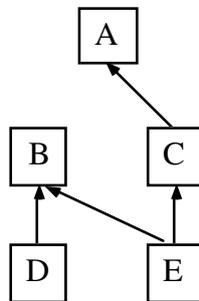


Figure 2. A class inheritance tree with multiple roots.

- *Example.* Consider the class inheritance diagram in Figure 1, $\text{NOC}(A) = \text{NOC}(B) = 2$ because the number of their immediate children are two. $\text{NOC}(C) = 1$ because the number of its immediate child is one. $\text{NOC}(D) = \text{NOC}(E) = 0$ because they have no children.

DIT indicates the extent to which the class is influenced by the properties of its ancestors and NOC indicates the potential impact on the descendants. CK argue that depth is preferred to breadth in the hierarchy.

2.2. Li's NAC and NDC definitions

Li [12] pointed out that CK's DIT metric has ambiguities. First, the definition of DIT is ambiguous when multiple inheritance and multiple roots are present at the same time. Consider the class inheritance tree with multiple roots in Figure 2, the maximum length from class E becomes unclear. There are two roots in this design, the maximum length from class E to root B is one ($\text{DIT}(E) = 1$) and the maximum length from class E to root A is two ($\text{DIT}(E) = 2$).

The second factor, which causes ambiguity, lies in the conflicting goals stated in the definition and the theoretical basis for the DIT metric. The theoretical basis stated that 'DIT is a measure of how many ancestor classes can potentially affect this class'. It is seen and understood to indicate that the DIT metric should measure the number of ancestor classes of a class. However, the definition of DIT stated that it should measure the length of the path in the inheritance tree, which is the distance between two nodes in a graph; this clearly conflicts with the measurement attribute declared in the theoretical basis. This ambiguity is only visible when multiple inheritances are present, where the distance between a class and the root class in the inheritance tree no longer yields the same number as the number of ancestor classes for the class. This conflict is shown in Figure 1 where, according to the definition, classes D and E have the same maximum length from the root of the tree to the nodes respectively; thus $\text{DIT}(D) = \text{DIT}(E) = 2$. However, class E inherits from more classes than D. According to the theoretical basis, class D and E should have different DIT values.

Li proposed a new metric, the Number of Ancestor Classes (NAC), as an alternative to the DIT metric.

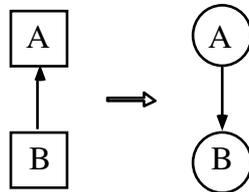


Figure 3. Notation for class inheritance hierarchy.

- *Definition.* NAC measures the total number of ancestor classes from which a class inherits in the class inheritance hierarchy.

Based on the definition, there exists no ambiguity on the NAC values for classes D and E in both Figures 1 and 2. In Figure 1, class D inherits from classes B and A, therefore yielding $NAC(D) = 2$; class E inherits from classes B, C and A, thus yielding $NAC(E) = 3$. In Figure 2, class D inherits from class B, thus yielding $NAC(D) = 1$; class E inherits from classes B, C and A, therefore yielding $NAC(E) = 3$.

Li [12] also points out that CK's NOC metric has some ambiguities. The stated theoretical basis and the viewpoints indicate that the NOC metric measures the scope of the influence of a class on its subclasses because of inheritance. It is not clear why only the immediate subclasses of a class are counted because a class has influence over all its subclasses, immediate or non-immediate. To remedy this insufficiency, Li proposed a new metric, the Number of Descendent Classes (NDC) metric, as an alternative to the NOC metric.

- *Definition.* The NDC metric is the total number of descendent classes (subclasses) of a class.

Consider the class inheritance tree in Figure 1, $NDC(A) = 4$, $NDC(B) = 2$ and $NDC(C) = 1$. In Figure 2, $NDC(A) = NDC(B) = 2$.

Inheritance helps reuse of already-designed classes when designing a new class. However, it also introduces coupling among the classes. If a change is made in the ancestor class, the change could potentially affect all its descendent classes. Therefore, the NAC and NDC metrics are important in measuring potential change propagation through the inheritance hierarchy [15]. We extend Li's metrics to specifically apply to the maintenance of class inheritance hierarchy.

3. METRICS FOR MAINTAINABILITY

To begin, we change the notation of the class inheritance tree. The term class inheritance tree is not valid, because if it has multiple inheritance it is not a tree but a graph. The most suitable mathematical model for describing an object taxonomy with inheritances is a directed acyclic graph (DAG) with no loops [20]. We change, therefore, the notation of class inheritance tree to class inheritance DAG as in Figure 3.

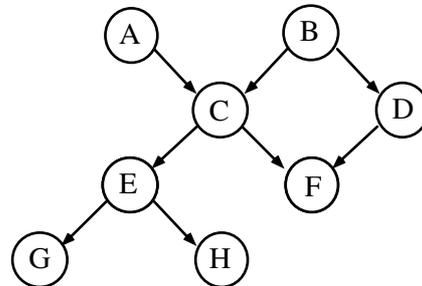


Figure 4. A class inheritance DAG.

For the definition of metrics for maintainability, we will use the terms from graph theory. In the directed graph, such as Figure 4, whose vertices represent the activities and edges represent the preceding relationship, vertex i is a predecessor of vertex j if there exists a path from vertex i to vertex j , and vertex j is a successor of vertex i [21]. We define functions $PRED$ and $SUCC$ as follows:

- $PRED(i)$: the total number of predecessors of node i ,
- $SUCC(i)$: the total number of successors of node i ,

For example in Figure 4, $PRED(E) = 3$, $PRED(A) = PRED(B) = 0$, $SUCC(C) = 4$, $SUCC(G) = SUCC(H) = 0$.

There are two activities in maintenance, understanding the structure of the system and modifying the system. Understandability is defined as the ease of understanding a program structure or a class inheritance structure and modifiability is defined as the ease with which a change or changes can be made to a program structure or a class inheritance structure. Consequently, we suggest a metric for understandability and a metric for modifiability as constituting the metrics for maintainability.

The measurement model of Kriz, promoted by Zuse in [22], shows that measurement includes empirical statements and results. Zuse claims that, for the most part, we *use* empirical statements and conditions and *do not use* formal mathematical conditions. Weyuker [9] uses this notion to characterize the properties of software measurement. Indeed, some scientists argue that a (good) software measure is a function that includes all the information needed and that it is not necessary to consider the empirical world (complicated by human aspects). Moreover, it is true that all the information and interpretations are in the measures or in the numerical properties. However, software engineering is without a doubt an empirical science, having on one side an empirical science (among others) and on the other side measures as mathematical functions, presenting the need to conciliate. In general terms, the majority of recommended properties are qualitative in nature and, consequently, most proposals for metrics have tended to be informal in their evaluation of metrics [8].

Furthermore, programming is sometimes called an art. In this sense, programming may be viewed as a technique complete with heuristics such as with art. It follows from this that software metrics should consider heuristic properties. Accordingly, it is not reasonable to measure the software product and process, which is actually a labor-intensive industry, only by mathematical and logical metrics without



considering the human aspects. Therefore, we shall consider the measurement of understandability and modifiability in a heuristic way[¶].

3.1. Metric of understandability

Ideally, objects should be independent, which makes it easy to transplant an object from one environment to another and to reuse existing objects when building new systems. However, the reality is that objects *do have* interdependencies and reusing them is rarely as simple as cutting and pasting [23]. The number of dependencies is typically so large that understanding and modifying the objects is not very easy.

In Figure 4, if we want to understand the content of class F, we should heuristically not only read class F, but also read classes C and D, which are superclasses due to inheritance. Additionally, we should also read classes A and B to understand C because they are superclasses of class C. As a result, we should understand five classes, F, C, D, A, B. This value is $PRED(F) + 1$. Consequently, we define the degree of understandability (U) of a class as follows:

$$U \text{ of class } C_i = PRED(C_i) + 1 \quad (1)$$

where C_i is i th class. The total degree of understandability (TU) of a class inheritance DAG is defined as follows:

$$TU \text{ of a class inheritance DAG} = \sum_{i=1}^t (PRED(C_i) + 1) \quad (2)$$

where t is the total number of classes in the class inheritance DAG.

When we talk about the understandability or complexity of a program there are many factors that may be considered. Properties such as timing correctness and synchronization are some examples of factors that affect the degree of understandability. As a matter of direct observation, large programs are generally (but not necessarily) more complex than small programs. Accordingly, it is more reasonable to compare the complexity of a program (or portion there of) to the complexity of another program of the same size. In other words, we should introduce the concept of averages. It is the same for the class inheritance DAG. The average depth of inheritance indicates the general level of modeling or abstraction used in the hierarchy [18]. Therefore, we define the average degree of understandability (AU) of a class inheritance DAG as follows.

$$AU \text{ of a class inheritance DAG} = \left(\sum_{i=1}^t (PRED(C_i) + 1) \right) / t \quad (3)$$

For example in Figure 4, by expression (1),

$$U(A) = U(B) = 1, \quad U(C) = 3, \quad U(D) = 2, \quad U(E) = 4, \\ U(F) = 5 \quad \text{and} \quad U(G) = U(H) = 5$$

[¶]Heuristic: (1) involving or serving as an aid to learning, discovery or problem-solving by experimental and especially trial-and-error methods; (2) of or relating to exploratory problem-solving techniques that utilize self-educating techniques (as the evaluation of feedback) to improve performance (e.g. a *heuristic* computer program).



hence, by expression (2),

$$TU = 1 + 1 + 3 + 2 + 4 + 5 + 5 + 5 = 26$$

and, by expression (3),

$$AU = 26/8 = 3.25$$

This value represents the degree of understandability, as defined above, for the class inheritance hierarchy of Figure 4. Naturally, as we have pointed out above (beginning of Section 3), understandability is a property that must be validated initially in a heuristic way and then empirically. At this time, we argue that from a heuristic point of view that this connotation is appropriate and later (in Sections 5 and 6) acknowledge there is a need for further experimentation to indeed show the correlation between our AU metric and the concept of understandability.

The value of AU above, is the same as the mean value of Li's NAC metric plus one (see Section 2.2 and Figure 2). An important point to emphasize here is that a lower value of AU highlights better understandability, since the figure actually reflects the degree of interdependence with other classes. When understanding a class, the less interdependencies there are, the easier will it be to understand. The value of AU is defined to provide completeness to our metrics and, subsequently, the metric of AM (average modifiability) is defined on the basis of AU in the next section.

3.2. Metric of modifiability

In Figure 4, if we want to modify the content of the class C, we should, first, understand what is the $U(C)$ value of U and only then modify class C. If class C affects subclass E and/or F, we should understand them to modify class C. Also, if the class E affects subclass G and/or H, we should understand them to modify class E. In the worst case, all the successor subclasses of class C must be modified! Naturally, in the best case, only class C will need to be modified. *It follows that half of the successor subclasses should be modified on average.* The modifiability of class C, therefore, could be $U(C) + SUCC(C)/2$. Consequently, we define the degree of modifiability (M) of a class as follows:

$$M \text{ of a class } C_i = U(C_i) + SUCC(C_i)/2 \quad (4)$$

where C_i is the i th class. The total degree of modifiability (TM) of a class inheritance DAG is as follows:

$$TM \text{ of the class inheritance DAG} = TU + \sum_{i=1}^t (SUCC(C_i)/2) \quad (5)$$

where t is the total number of classes in the class inheritance DAG.

The average degree of modifiability (AM) of a class inheritance DAG is as follows:

$$AM \text{ of the class inheritance DAG} = AU + \left(\sum_{i=1}^t (SUCC(C_i)/2) \right) / t \quad (6)$$



for example in Figure 4, by expression (4),

$$M(A) = U(A) + SUCC(A)/2 = 1 + 5/2 = 3.5$$

$$M(B) = U(B) + SUCC(B)/2 = 1 + 6/2 = 4$$

$$M(C) = U(C) + SUCC(C)/2 = 3 + 4/2 = 5$$

$$M(D) = U(D) + SUCC(D)/2 = 2 + 1/2 = 2.5$$

$$M(E) = U(E) + SUCC(E)/2 = 4 + 2/2 = 5$$

$$M(F) = U(F) + SUCC(F)/2 = 5 + 0/2 = 5$$

$$M(G) = U(G) + SUCC(G)/2 = 5 + 0/2 = 5$$

$$M(H) = U(H) + SUCC(H)/2 = 5 + 0/2 = 5$$

therefore, by expression (5),

$$TM = 3.5 + 4 + 5 + 2.5 + 5 + 5 + 5 + 5 = 35$$

and, by expression (6)

$$AM = 35/8 = 4.38$$

This value represents the degree of modifiability of the class inheritance hierarchy in Figure 4. Similarly, as pointed out in the case of average understandability in the previous section, a lower value of AM represents a better index for modifiability. CK's DIT of Figure 4 is three. Henderson-Sellers [18] recommended seven as a rough guideline for maximum DIT^{||}.

Coad's one heuristic recommendation is: 'Avoid having too many services per class. Each class typically has no more than six or seven public services' [24]. Therefore, we recommend that the maximum value of AM of the class inheritance hierarchy be under seven. Heuristically, we recommend that the best value of AM is four, the median of one through seven (as a general, yet speculative, guideline).

4. DISCUSSION

We discuss our metrics and compare them with CK's DIT and Henderson-Sellers's AID.

4.1. CK's DIT

CK [7] proposed that it is better to have depth rather than breadth in the inheritance hierarchy so that the NOC measure, which counts the number of immediate subclasses, has larger values for poorer designs. In contrast, from the maintenance viewpoint, this is not reasonable. For example in Figure 5, CK prefers class inheritance hierarchy (a) to (b). Yet, from the heuristic point of view, Figure 5(b) is usually easier to understand than Figure 5(a). Arguably, the hierarchy of Figure 5(a) is straightforward compared with 5(b) yet, when this general trend is extrapolated, the premise that 5(b) should be preferred becomes

^{||}Research has shown that people remember at best seven things at one time.

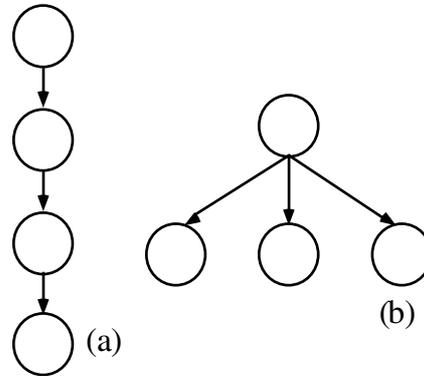


Figure 5. Two class inheritance DAGs.

much stronger from the heuristic viewpoint:

$$AU \text{ of Figure 5(a): } (1 + 2 + 3 + 4)/4 = 10/4 = 2.5$$

$$AM \text{ of Figure 5(a): } 2.5 + (3/2 + 2/2 + 1/2)/4 = 3.25$$

$$AU \text{ of Figure 5(b): } (1 + 2 + 2 + 2)/4 = 7/4 = 1.75$$

$$AM \text{ of Figure 5(b): } 1.75 + (3/2)/4 = 2.13$$

The understandability and modifiability of Figure 5(b) is better than Figure 5(a) in our metrics.

4.2. Henderson-Sellers's AID

The AID of a class is calculated by [18] as:

$$\left(\sum \text{depth of each class} \right) / \text{number of classes}$$

For example, consider two class inheritance DAGs in Figure 6:

$$AID \text{ of Figure 6(a): } (0 + 0 + 1 + 1 + (1 + 2)/2)/5 = 0.7$$

$$AID \text{ of Figure 6(b): } (0 + 1 + 1 + 1)/4 = 0.75$$

Figure 6(b) is definitely easier to understand than Figure 6(a), but the AID values for them are just the reverse. Calculations of the AU and AM are as follows.

$$AU \text{ of Figure 6(a): } (1 + 1 + 2 + 2 + 4)/5 = 2$$

$$AM \text{ of Figure 6(a): } 2 + (2/2 + 2/2 + 1/2)/5 = 2.5$$

$$AU \text{ of Figure 6(b): } (1 + 2 + 2 + 2)/4 = 1.75$$

$$AM \text{ of Figure 6(b): } 1.75 + (3/2)/4 = 2.13$$

The understandability and modifiability of Figure 6(b) is better than Figure 6(a) in our metrics.

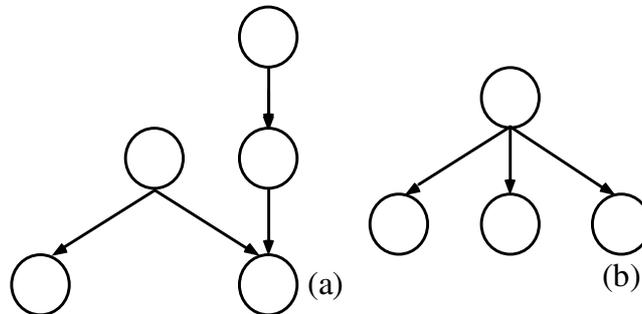


Figure 6. Another two class inheritance DAGs.

5. CONCLUSION

There is no single measure that captures all the features of an OO software product [2]. Based on this fact, we have proposed two simple and heuristically validated metrics to assess class inheritance hierarchy in terms of understandability and modifiability.

These metrics were developed in the light of an extensive review of the work of CK and Li. We have extended their metrics to apply specifically to maintenance of class inheritance hierarchy. We suggest new metrics for understandability and modifiability of class inheritance hierarchy. In addition, several comparisons have been made using two different design versions to give an illustrative example. The advantages over CK's metrics and Henderson-Sellers's metrics are explained in the context of maintenance for a class inheritance hierarchy. We have given several theoretical and intuitive arguments to support our claims in this regard. More empirical evidence is required before these new metrics can be claimed to work better in practice.

Our primary premise argues that the deeper the hierarchy of an inheritance tree, the better it is for reusability, but worse for maintenance (i.e. detrimental results can be expected). The shallower the hierarchy, the less the abstraction, but the better the understanding and modifying. Taking the maintenance point of view, it is recommended that a deep inheritance tree should be split into a shallow inheritance tree. In this way, the benefit of using the *AU* and *AM* metrics is the added insight gained about trade-offs among conflicting requirements that promote increased reuse (via inheritance) and ease of maintenance (via a less complicated inheritance hierarchy). Both designers and managers can benefit from the use of these proposed metrics: (1) the former from checking the quality of their work and as a guide for improvement and (2) the latter to estimate, schedule and control the design and maintenance activities for the project.

6. FUTURE WORK

The *AU* and *AM* metrics were developed for application during the design phase of the lifecycle to measure the maintainability of a class inheritance hierarchy. The metrics assume that the complexity of all the classes are the same value (i.e. one). These metrics can be extended to comprehend the complexity of a class inheritance hierarchy. For example, one approach might look as the following.



A class is composed of attributes and methods. The complexity of a class is, therefore, the sum of the complexity of attributes and that of the methods [25–27]. Thus, the complexity of C_i could be expressed as follows.

$$\text{Complexity of } C_i = \text{attr}(C_i) + \text{method}(C_i)$$

where:

$\text{attr}(C_i)$ = sum of the weighted attributes depending on variable, array, structure, etc. in the class C_i ; and

$\text{method}(C_i)$ = sum of cyclomatic number of each method in the class C_i .

Therefore, in terms of future research, we plan to study some fundamental issues. We will investigate (1) extending our current set of metrics by addressing how to strengthen the inherent assumptions and (2) to determine whether there is an optimum level of inheritance for reducing the maintenance efforts while at the same time preserving the potential for reuse. From this cause, our next study is to extend and evaluate our metrics empirically by using real data.

REFERENCES

1. Balasubramanian NV. Object oriented metrics. *Proceedings 3rd Asia-Pacific Software Engineering Conference (APSEC'96)*. IEEE Computer Society Press: Los Alamitos CA, 1996; 30–34.
2. Tahvildari L, Singh A. Categorization of object-oriented software metrics. *Proceedings IEEE Canadian Conference on Electrical and Computer Engineering*. IEEE Computer Society Press: Los Alamitos CA, 2000; 235–239.
3. Tegarden DP, Sheetz SD, Monarchi DE. A software complexity model of object-oriented systems. *Decision Support Systems* 1995; **13**(3–4):241–262.
4. McCabe TJ, Dreyer LA, Dunn AJ, Watson AH. *Testing an Object-Oriented Application*. Quality Insurance Institute: Orlando FL, 1994; 21–27.
5. Harrison R, Counsell SJ, Nithi RV. An evaluation of the MOOD set of object-oriented software metrics. *IEEE Transactions on Software Engineering* 1998; **24**(6):491–496.
6. Briand LC, Morasoa S. Defining and validating measures for object-based high level design. *IEEE Transactions on Software Engineering* 1999; **25**(5):722–743.
7. Chidamber SR, Kemerer CF. Towards a metric suite for object-oriented design. *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*. ACM Press: New York NY, 1991; 197–211.
8. Chidamber SR, Kemerer CF. A metrics suite for object-oriented design. *IEEE Transactions on Software Engineering* 1994; **20**(6):476–493.
9. Weyuker EJ. Evaluating software complexity measures. *IEEE Transactions on Software Engineering* 1988; **14**(9):1357–1365.
10. Basili VR, Briand LC, Melo WL. A validation of object-oriented design metrics as quality indicators. *Technical Report*, University of Maryland, Department of Computer Science, College Park MD, 1995; 1–24.
11. Tang MH, Kao MH, Chen MH. An empirical study on object-oriented metrics. *Proceedings 23rd Annual International Computer Software and Application Conference*. IEEE Computer Society Press: Los Alamitos CA, 1999; 242–249.
12. Li W. Another metric suite for object-oriented programming. *The Journal of Systems and Software* 1998; **44**(2):155–162.
13. Kitchenham B, Pflieger SL, Fenton NE. Towards a framework for software measurement validation. *IEEE Transactions on Software Engineering* 1995; **21**(12):929–944.
14. Fenton NE, Neil M. Software metrics: Successes, failures and new directions. *The Journal of Systems and Software* 1999; **47**(2–3):149–157.
15. Harrison R, Counsell SJ, Nithi RV. An evaluation of the MOOD set of object-oriented software metrics. *IEEE Transactions on Software Engineering* 1998; **24**(6):491–496.
16. Daly J, Brooks A, Miller J, Roper M, Wood M. Evaluating inheritance depth on the maintainability of object-oriented software. *Empirical Software Engineering* 1996; **1**(2):109–132.
17. Briand L, Wust J, Daly J, Porter V. Exploring the relationships between design measures and software quality on object-oriented systems. *The Journal of Systems and Software* 2000; **51**(3):245–273.



18. Henderson-Sellers B. *Object Oriented Metrics: Measures of Complexity*. Prentice Hall PTR: Englewood Cliffs NJ, 1996; 130–132.
19. Lake A, Cook C. Use of factor analysis to develop OOP software complexity metrics. *Proceedings of the Annual Oregon Workshop on Software Metrics*. Oregon Center for Advanced Technology: Portland OR, 1994. <http://www.ocate.edu/> [8 December 2001].
20. Wang CC, Shih TK, Pai WC. An automatic approach to object-oriented software testing and metrics for C++ inheritance hierarchies. *Proceedings International Conference on Automated Software Engineering (ASE'97)*. IEEE Computer Society Press: Los Alamitos CA, 1997; 934–938.
21. Horowitz E, Sahni S. *Fundamentals of Data Structures in C*. Computer Science Press: New York, 1993; 303–304.
22. Zuse H. *A Framework of Software Measurement*. Walter de Gruyter: Berlin, 1997; 103–114.
23. Schroeder M. A practical guide to object oriented metrics. *IT Pro* 1999; 1(6):30–36.
24. Coad P. OOD Criteria, Part 3. *Journal of Object Oriented Computing* 1991; 4(5):67–70.
25. Kafura DG, Reddy GR. The use of software complexity metrics in software maintenance. *IEEE Transactions on Software Engineering* 1987; SE-13(3):335–343.
26. Navlakha JK. A survey of system complexity metrics. *The Computer Journal* 1987; 30(3):233–238.
27. Cha SD. Information theoretical complexity metrics for concurrent programs. *Doctoral Dissertation*, Korea Advanced Institute of Science and Technology (KAIST), Seoul, Korea, 1995; 118 pp.

AUTHORS' BIOGRAPHIES



Frederick T. Sheldon is an Assistant Professor at the Washington State University teaching and conducting research in the area of software engineering, developing and validating methods and tools for creation of safe and correct software (including software evolution in the area of extensibility). He is currently on leave at DaimlerChrysler Research and Technology in Stuttgart. Dr Sheldon received his PhD at the University of Texas at Arlington (UTA) and has worked at NASA Langley and Ames Research Centers, General Dynamics (Lockheed Martin) and Texas Instruments (Raytheon).



Kshamta Jerath is a Graduate Student at the Washington State University pursuing a master's degree in Computer Science. Her research interests include Software Engineering, Safety and Reliability Analysis. Ms. Jerath was working as a software engineer with IBM India until December 2000. She holds a bachelor's degree in Computer Engineering from Delhi College of Engineering, India.



Hong Chung is an Associate Professor at the Keimyung University, Korea. His research interests include Object-Oriented Software Metrics, Software Design Methodology and Data Mining. Dr Chung received his PhD at the Catholic University at Taegu, Korea. He also holds an MBA in Production Control from Korea University. Dr Chung has worked as an Instructor at Sogang University and as a Senior Researcher, Computer Laboratory, Korea Institute of Science and Technology, Seoul.