
Research

Measuring the complexity of class diagrams in reverse engineering



Frederick T. Sheldon^{1,*} and Hong Chung²

¹*Computational Science and Engineering, Oak Ridge National Laboratory, Oak Ridge, TN 37831, U.S.A.*

²*Department of Computer Science, Keimyung University, Taegu 704-701, South Korea*

SUMMARY

Complexity metrics for object-oriented systems are plentiful. Numerous studies have been undertaken to establish valid and meaningful measures of maintainability as they relate to the static structural characteristics of software. In general, these studies have lacked the empirical validation of their meaning and/or have only succeeded in evaluating partial aspects of the system. In this study we have determined, through limited empirical means, a practical and holistic view by analyzing and comparing the structural characteristics of UML class diagrams as those characteristics relate to or impact maintainability. Class diagrams are composed of three kinds of relation, association, generalization, and aggregation, which make their overall structure difficult to understand. We propose combining these three relations in such a way that enables a comprehensive measure of complexity. Theoretically, this measure is applicable among different class diagrams (including different domains, platforms or systems) to the extent that the measure is widely comparative and context free. Further, this property does not preclude comparison within a specific class diagram (or family) and is therefore very useful in evaluating a given class diagram's strengths and weaknesses. We are reporting empirical results that provide a small measure of validity to enable an objective appraisal of both complexity and maintainability without equating the two. Therefore, to evaluate our structural complexity metric, we determined the level of understandability of the system by measuring the time needed to reverse engineer source code into class diagrams including the number of errors produced while creating the diagram. The number of errors produced offers one indicator of maintainability. The results, as compared with other complexity metrics, indicate that our metric shows promise especially if proven to be scalable. Copyright © 2006 John Wiley & Sons, Ltd.

Received 21 July 2005; Revised 8 August 2006; Accepted 10 August 2006

KEY WORDS: software quality; perfective/corrective maintenance; coupling/cohesion; object-oriented complexity metrics; class inheritance and diagrams

*Correspondence to: Frederick Sheldon, Oak Ridge National Laboratory, Oak Ridge, TN 37831, U.S.A.

†E-mail: sheldonft@ornl.gov

Contract/grant sponsor: U.S. Department of Energy (DOE); contract/grant number: DE-AC05-00OR22725

Contract/grant sponsor: Keimyung University, South Korea



1. INTRODUCTION

Over the past two decades, the software industry has moved to adopt the object-oriented (OO) paradigm from a myriad of structured programming paradigms. Numerous studies have been conducted that provide both theoretical [1–6] and empirical [7–15] rationale to support this paradigm shift [15,16]. The common thread among most of these studies has been the measurement of static structure along with some sort of validation of the proposed metrics. In most of these studies, as is the case for this work, the root concept deals with the complexity of OO systems. Examples include: the research of Chidamber and Kemerer (C&K) [3,17], who proposed six OO metrics (CK metrics) which are now often used to measure the complexity of OO systems; Li's [5] proposed metrics that were aimed at overcoming perceived deficiencies found in the CK metrics; Sheldon *et al.*'s [6] suggested metrics based on Li's studies which measure both the understandability and modifiability of inheritance hierarchies for the purpose of maintaining such structures; Abreu's [1] proposed metrics that address encapsulation, inheritance, coupling, and polymorphism of OO systems backed by experimental validation; Briand *et al.*'s [2] suggestion of a unified framework for coupling measurement in OO systems; and Ferneley's [4] proposed coupling and control flow measures. Almost all of these metrics endeavor to measure some partial aspect of the system(s) under study. Meanwhile, practitioners clamor for a unified metric to measure the overall system complexity that is both simple to use and easy to understand from a cognitive point of view [18–22].

In this paper, we present an analysis of class diagrams used in the design of static structures by means of UML (Unified Modeling Language). UML provides standardized OO notation for specifying both the static and dynamic characteristics of OO systems and is a natural foundation for assessing overall system complexity. By measuring the complexity of a system during the design phase, it may be possible to avoid higher complexity designs, possibly reduce development cost, and facilitate the improvement of tasks during the implementation, maintenance, and evolutionary phases. In this way, such measures can help to refine architectural approaches. Therefore, in evaluating the complexity of OO system structures, we strived to evaluate the level of understandability. In doing so, we measured (1) the time needed to reverse engineer the source code into a class diagram, and (2) the number of errors that occurred during the process of creating the diagram. Our results indicate that the proposed metric shows a considerably strong relationship with the complexity of the OO system.

Section 2 highlights metrics suggested by C&K [3], Li [5], and Sheldon *et al.* [6] as the ground work for our research. Three types of class relation, association, generalization, and aggregation, are detailed in Section 3, while Section 4 identifies the key functions of a university information system for which we present the class diagrams used to exemplify and semi-validate our new OO complexity metric.

2. RELATED WORK

C&K [3] proposed six metrics for the measurement of OO systems. DIT (Depth of Inheritance Tree) measures the depth of a class inheritance tree (i.e., defined as the depth of inheritance for a class) by counting the number of ancestor classes that can affect a given class. NOC (Number of Children) measures the width of a class inheritance tree (i.e., defined as the number of immediate subclasses subordinate to a class) by counting the number of subclasses that inherit the methods of a parent class. WMC (Weighted Methods per Class) counts the number of methods in a class and is defined as the



sum of the complexity of the class' local methods. C&K suggest that the complexity of a method can be measured using LOC (Lines of Codes) or McCabe's CC (Cyclomatic Complexity). We cannot count the LOC or CC at the early phase of design and therefore must use the WMC value as the number of methods. RFC (Response for Class) is defined as the cardinality of a set of methods that can potentially be executed in response to a message received by an object of that class. CBO (Coupling Between Objects) is defined as a count of the number of other classes to which it is coupled. LCOM (Lack of Cohesion in Methods) measures the inter-relatedness between portions (i.e., objects, modules, disparate code segments) of a program and is defined as the count of the number of method pairs whose similarity is 0 (zero), minus the count of method pairs whose similarity is not zero. DIT and NOC are for the measurement of class inheritance structures, WMC, RFC and LCOM are for measuring class complexity, and CBO estimates the coupling relationship among classes.

Li [5] has proposed six metrics concerned with the complexity of OO systems. NAC (Number of Ancestor Classes) measures the complexity of a class inheritance hierarchy and is defined as the total number of ancestor (predecessor) classes from the current class. Consider the class inheritance hierarchy as represented by a directed graph (i.e., a rooted tree). The NAC metric counts the number of nodes reachable from a node within the tree (i.e., a class), or the number of all ancestor classes that affect that particular class. The NDC (Number of Descendent Classes) metric also measures the complexity of a class inheritance hierarchy and is defined as the total number of descendent classes (subclasses) of a class. The metric counts the number of nodes reachable from a class, and considers all descendent classes that affect that particular predecessor class. NLM (Number of Local Methods) is defined as the number of the local methods defined in a class that is accessible outside the class (local methods are the public methods of C++/Java). This metric captures the size of a class' local interface through which other classes can use the class. CMC (Class Method Complexity) is defined as the summation of the internal structural complexity of all local methods, regardless of whether they are visible outside the class or not. The local methods are all of the public and private methods. The structural complexity can be measured using LOC or CC. CTA (Coupling Through Abstract data types) measures the coupling between classes and is defined as the total number of classes that are used as abstract data types in the data-attribute declaration of a class. This metric gives the scope of how many other class services a class needs to provide as its own service to others. Yet another metric, the CTM (Coupling Through Message passing), captures the dynamic coupling between objects because it counts the number of different messages sent from a class to other classes, excluding the messages sent to the objects created as local objects in the local methods of the class. This metric gives an indication of how many methods (services) from other classes are needed to fulfill the class' own functionality. Among the six metrics discussed above, NAC and NDC measure class inheritance structure, NLM and CMC measure class method complexity, while CTA and CTM are for determining the strength of coupling between classes.

Measuring coupling is a means of appraising the connectedness of a software module in a structure of software modules (a system implemented with computer software) in an environment. The connectedness depends on the character and content of the data created or communicated by a module or the environment, and subsequently used in some way by some other module or modules in the same structure of software modules (a system implemented with computer software) or by the environment. The connectedness does not depend on the frequency or volume of the data flows between the modules or to and from the environment, but does depend on the content, number, and variety of the messages [23]. Clearly, structural complexity and coupling are not equivalent. Coupling can be used to ascertain and describe the connectedness of the design and/or code.



Sheldon *et al.* [6] suggested two kinds of metric for measuring the complexity of class inheritance structures. The first is U (Understandability) for understanding the class inheritance hierarchy and is defined as the total number of ancestor classes which affect a class. If we represent the class inheritance tree as a Directed Acyclic Graph (DAG), the metric counts the number of all predecessor nodes. The second kind of metric is M (Modifiability), which measures the modifiability of class inheritance hierarchies and is defined as the total number of descendent classes that are affected by a class. The metric counts the number of all successor nodes and is then divided by two (i.e., half of the successor subclasses would be modified on average). On this basis, they proposed TU (Total Understandability) and TM (Total Modifiability), which are summations of each U and M for the total number of classes in the class inheritance tree. They also suggested AU (Average Understandability) and AM (Average Modifiability) to give the average of each TM and TU divided by the total number of classes. Both metrics give an indication of the complexity of OO class inheritance structures.

A class diagram consists of many classes and various kinds of relations among them. Therefore, the complexity of the diagram should be measured considering these relationships collectively. There are three types of relationship: (1) association, (2) generalization, and (3) aggregation.

The metrics reviewed here are concerned with OO system complexity and deal with only one part or aspect of the system structure. For example, in the metrics proposed by C&K [3], the generalization relationship is measured by DIT and NOC and the association relationship is measured by CBO. Also, in the metrics suggested by Li [5], the generalization relationship is measured by NAC and NDC and the association relationship is measured by CTA and CTM.

The true complexity of a system should be a composite picture considered in closer connection with the various parts of the structure. In this paper we investigate how to measure the class diagram in a way that includes all of these types of relation. Further, we will suggest a unified metric considering all relationships for measuring the complexity of relations among the classes. This new metric, which unifies DIT, NOC, and CBO, or CTA, CTM, NAC, and NDC, is proposed and semi-validated.

3. CLASS DIAGRAM COMPLEXITY ANALYSIS

The complexity of static structures represented by a class diagram is dependent on how the relationships between the classes are structured. Using the UML notation, an association relationship is represented by an arc connecting the associated classes, including unidirectional (one open arrow head) and bidirectional (no arrow head) links, as shown in Figure 1. An aggregation relationship is an association with a diamond next to the class denoting the aggregate, and a generalization relationship is an association with a closed arrowhead next to the class denoting the generalization.

3.1. Association relationship

The association relationship is the most widely used semantic connection between classes. This relationship asserts that a class knows the opposite class, so the class can send message(s) to the other classes to perform services (i.e., invoke a method). There are two types of associations, the unidirectional relationship where only one side knows the other, and the bidirectional relationship where both know each other. For example, the coding of a unidirectional association between the class Person and the class Computer in Figure 1 is given below.

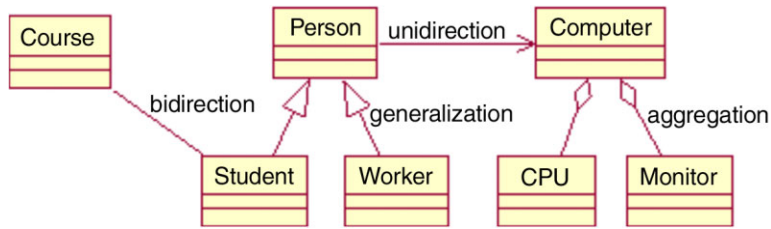


Figure 1. Simple class diagram used in the example descriptions.

```
class Person {
    private Computer theComputer;
    private int value;
    .....
    public string CalculateValue() {
        value = theComputer.Calculate();
    }
    .....
}
class Computer {
    .....
    public int Calculate() {...}
    .....
}
```

In this code, the class `Person` declares an object `theComputer`, which is an instance of the class `Computer`, and sends a message `Calculate()` to the object. By declaring the object `theComputer` in the class `Person` we mean that the class knows the contents of the class `Computer`. In this class diagram, only the class `Person` can send a message to the object `theComputer` because it knows (and can reference) the class `Computer` (but not *vice versa*). Now, let us consider the bidirectional association between the class `Person` and the class `Computer` from Figure 1 is given below.

```
class Course {
    private string courseId;
    private int credit;
    private int noOfSt;
    private Student theStudent;
    public string NumberOfStudent() {
        noOfSt = theStudent.GetNoOfSt();
    }
    public int GetCredit() {...}
    .....
}
```



```

class Student {
    private string id;
    private int name;
    private Course theCourse:
    private int totalCredit:
    public int GetNoOfSt() {...}
    public int CalculateCredit() {
        totalCredit = theCourse.GetCredit():
    }
    .....
}

```

In this code, the class Course declares an object theStudent, which is an instance of the class Student, and sends a message GetNoOfSt() to the object. The class Student declares an object theCourse, which is an instance of the class Course, and sends a message GetCredit() to the object. Declaring the objects within each class means that each class knows (and can reference) the contents of the opposite class. In the diagram, both classes can send messages to the opposite object because they know each other. The unidirectional relationship compared with the bidirectional relationship, in the aspect of complexity, is heuristically twice as complex because the class with unidirectional relationship knows only the structures of the opposite class, but the classes with the bidirectional relationship know each other.

3.2. Generalization relationship

The generalization relationship shows generalization/specialization through the concept of inheritance between classes. It is called the inheritance relationship because the subclasses inherit attributes and operations from the superclass. The superclass generalizes the common attributes of subclasses. A subclass is a specialized class, which inherits attributes from a superclass as well as adding new attributes, or redefining some inherited attributes. The coding examples of the superclass Person and the subclasses Student and Worker of Figure 1 are given below.

```

class Person {
    private string name:
    private int age:
    public string GetName() {...}
    Public int HowOld() {...}
}
class Student : Person {
    private string school:
    private int grade:
    public string WhatSchool() {...}
    public int WhatGrade() {...}
}
class Worker : person {
    private string company:

```



```
private int salary:
public string WhatCompany1() {...}
public int GetSalary() {...}
}
```

In these code segments, the class `Person` has common attributes and operations that the classes `Student` and `Worker` commonly possess. The classes `Student` and `Worker` inherit all of the attributes and operations from the superclass `Person` and additionally declare some new attributes and operations.

The inheritance relationship provides various useful concepts including reuse: reducing duplicate coding, by way of reusing all of the attributes and operations of the superclass. However, this relationship introduces complexity (by way of indirection) because to understand the subclass (e.g., `Student` or `Worker`), the superclass and its attributes (e.g., `Person`) must be understood. Therefore, the complexity of the inheritance structures should be measured as an attribute of understandability. The more difficult the structures are to understand, the more complex the general relationship. We use the metrics of understandability *U*, *TU*, and *AU* suggested by Sheldon *et al.* [6] to determine the complexity of the generalization relationship.

3.3. Aggregation relationship

The aggregation relationship is a specialized way to describe how the whole is related to its parts. Aggregation is known as a containment relationship. Often, it is difficult to choose whether the aggregation or the association relationship should be used. The key difference being that aggregation includes all subclasses (e.g., the class `Computer` includes both classes `CPU` and `Monitor`) while the association does not. Also, association is symmetrical to the bidirectional relationship, while aggregation is not. The coding example of the aggregation among the classes `Computer`, `CPU`, and `Monitor` in Figure 1 is given below.

```
class Computer {
    private int memorySize:
    private CPU theCPU:
    private Monitor theMonitor
    .....
}
class CPU {...}
class Monitor {...}
```

Here, the class `Computer` references both classes `CPU` and `Monitor` and therefore must know both of the other classes. The aggregation relationship is considered only as a unidirectional association from the class structure viewpoint, which is naturally less complex than a bidirectional association.

4. MEASURING THE COMPLEXITY OF OO SYSTEMS STATIC STRUCTURES

OO systems static structure complexity is naturally measured by the number of connections between classes. C&K [3] proposed the metric CBO for measuring the number of connections between objects, and they assert that the more connections the structures have, the more complex the connections are.



Henderson-Sellers [14] uses fan-in/fan-out between classes as the complexity metric for OO systems. CBO measures only the number of connections between classes without considering the direction of connection, while the fan-in/fan-out method considers direction. Some would argue that the complexity of a bidirectional association is twice as difficult to understand as compared with a unidirectional association because both fan-in and fan-out functionality are included simultaneously. In this work, we interpret the complexity of a unidirectional relationship as having one of either the fan-in or the fan-out properties while the bidirectional relationship includes both.

4.1. Class diagram complexity metric

To fully analyze the complexity of systems we collectively consider association, generalization, and aggregation together in the class diagram, which, in turn, represent the static structure of OO systems. In this light, we define the complexity based on the number of static relationships between the classes of the class diagram (i.e., the number of connections between classes). The complexity of the association and aggregation relationship is counted as the number of direct connections; whereas the generalization relationship is counted as the number of all ancestor classes and descendent classes. Our metric definition is developed below.

- (1) The complexity of the bidirectional relationship is twice as complex as that of the unidirectional case. The unidirectional relationship knows only the other, so-called opposite class to which it is connected, while classes with a bidirectional relationship must know each other. Knowledge of the opposite class implies that an object of a class knows the contents of the opposite object, which is an instance of the opposite class (i.e., an object can send a message to another object only if it knows the contents of the other). Therefore, we define the complexity of the association relationship (AR) as follows:

$$\text{AR} = (\text{Number of unidirectional relationships}) + 2 \times (\text{Number of bidirectional relationships}) \quad (1)$$

- (2) We use AU suggested by Sheldon *et al.* [6] to determine the complexity of the generalization relationship. The internal complexity of each class in an inheritance tree is regarded as the unit value 1:

$$\text{AU} = \frac{\sum_{i=1}^n (\text{PRED}(C_i) + 1)}{n} \quad (2)$$

Here, n is the number of classes in the inheritance tree, and $\text{PRED}(C_i)$ is the number of predecessor classes of the i th class C_i , $i = 1, \dots, n$. To understand the class C_i , we have to understand all of the ancestor classes that affect the class as well as the class C_i itself. An ancestor is the predecessor when the inheritance tree is represented as a DAG. The average is computed because the complexity of the generalization relationship (defined above) for each class is different.

- (3) We assume that the complexity of the aggregation relationship is equal to that of a unidirectional association. The aggregation class is a type of containment for all of the member classes that constitute the class, therefore this class must know the member classes, but the member classes need not know the aggregation class. As a result, the complexity of the aggregation relationship (AGR) is defined as follows:

$$\text{AGR} = (\text{Number of aggregation relationships}) \quad (3)$$



- (4) Expressions (1), (2), and (3) have the additive property of metric theory because the units of the expressions are the number of connections between classes. So, complexity of class diagram (CCD) can be expressed as a composition of the three expressions:

$$CCD = AR + AU + AGR \quad (4)$$

By substituting for each expression, the complexity of the class diagram is defined as follows.

$$\begin{aligned} CCD = & ((\text{Number of unidirectional relationships}) + 2 \\ & \times (\text{Number of bidirectional relationships})) \\ & + \frac{\sum_{i=1}^n (\text{PRED}(C_i) + 1)}{n} + (\text{Number of aggregation relationships}) \end{aligned} \quad (5)$$

The complexity of the class diagram in Figure 1 is computed by expression (5) as follows:

$$CCD = 1 + 2 \times 1 + \frac{(1 + 2 + 2)}{3} + 2 = 6.67 \quad (6)$$

The complexity (expression (6)) of the class diagram in Figure 1 is very simple. In our judgment, a CCD below 30 should be considered relatively simple while one that is over 30 should be considered relatively complex. This judgment is based on the observations of students engaged in learning the OO programming paradigm (via C++/Java) at the university undergraduate/graduate level. This threshold value of 30 is the result of our empirical study described in the following section.

4.2. Complexity metric application

The Keimyung University Information System (IS), an OO system written in Java, was used for our experiment. The system includes the functions of Student Management, Lectures Taking Management, Professor Management, Curriculum Management, Staff Management, User Management, and so on. We selected the Student Management, Lectures Taking Management, and User Management functions as the system for experiment because the other functions generally encompass the same activities. We then divided the system into six subsystems as described in the next section.

Participants in the experiment included 24 undergraduate students who had taken the department's OO systems course with a grade of B (or better), and six graduate students. Six groups consisting of four undergraduates and one graduate were formed. System understandability was measured based on a carefully observed (via the graduate student assigned to each group) student group assessment using the following objective measures: time to draw the class diagram hierarchy (reverse engineered from the IS source code) and the number of errors discovered in the diagram.

Generally, undergraduate students work as junior programmers when they graduate from school, and graduate students work as senior programmers or system analysts and are regarded as experts. The junior programmers write program code well but they are unskilled in designing the structure of systems. In contrast, the senior programmers are very skilled in the design and structure of software systems. We had both levels of programmers participate in the experiment and utilized the average value of their measured values.

A small-scale programmer team or a subproject team generally consists of one senior programmer and three to five junior programmers. Furthermore, the results of our experiment, which included six distinct groups of one senior and four junior programmers provided some gauge of construct validity.

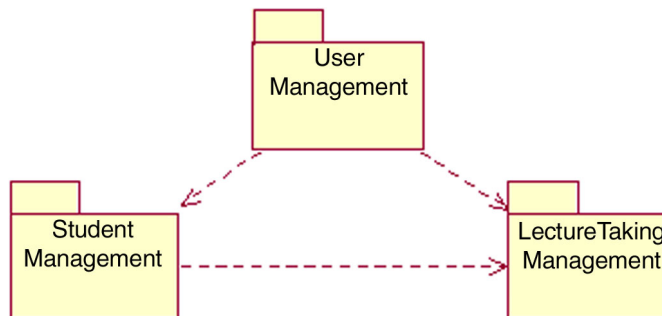


Figure 2. The Keimyung University IS package diagram.

Construct validity seeks agreement between a theoretical concept and a specific measuring device or procedure [24]. For example, for inventing a new measure of complexity we have attempted to ‘define’ maintainability by assessing how easily a structure can be reverse engineered, to facilitate a satisfactory level of construct validity. Construct validity is usually divided into subcategories: convergent validity and discriminate validity. Convergent validity is the actual general agreement among ratings, gathered independently of one another, where measures should be theoretically related. Discriminate validity is the lack of a relationship among measures, which theoretically should not be related. To understand whether a piece of research has construct validity, three steps should be followed. First, the theoretical relationships must be specified. Second, the empirical relationships between the measures of the concepts must be examined. Third, the empirical evidence must be interpreted in terms of how it clarifies the construct validity of the particular measure being tested [25, p. 23]. These last two steps outline our analysis (including [6]) toward assessing the validity of our metric, as described below.

4.2.1. Example system structures

The UML package diagram shown in Figure 2 represents the system structure. The UML class diagrams for the User Management, Student Management, and Lectures Taking Management functions are also shown in Figures 3, 4, and 5, respectively. We developed six different cases for the experiment: Case A, User Management; Case B, Student Management; Case C, Lectures Taking Management; Case D, User Management + Student Management; Case E, User Management + Lecture Taking Management; and Case F, User Management + Student Management + Lecture Taking Management. The complexity of the six cases, measured by our proposed metric, the CK metrics, and Li’s metrics, from the class diagrams in Figures 3, 4 and 5, are in Table I.

The values of CCD, CBO, and CTA in Table I are plotted in Figure 6. The values of CCD are a little larger than those of CBO and CTA. This is because the system used for the experiment has only one level of inheritance, and so the difference of values of the three metrics is small.

In general, most OO systems do not sufficiently utilize the inheritance property, which is a known good practice within the OO paradigm. For example, in the system with 180 classes used in the

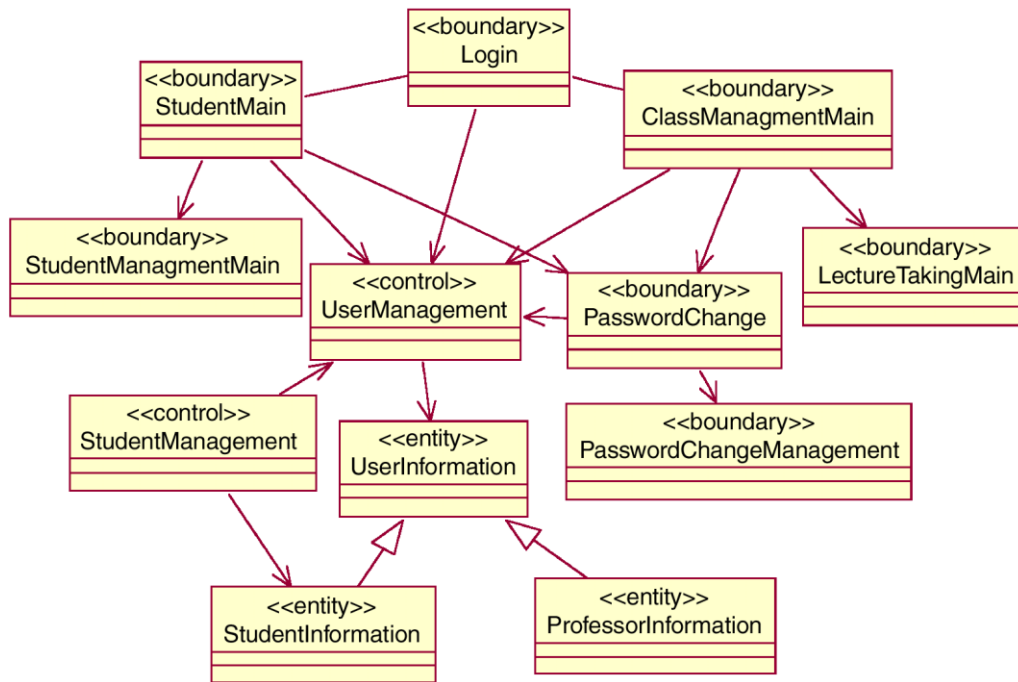


Figure 3. Class diagram for user management.

experiment of Basili *et al.* [7], CBO is 15, but DIT is 3 and NOC is only 2, and in the system including 27 classes used in the experiment of Chidamber *et al.* [26], CBO is 22, but DIT and NOC are 2. Similarly, our system has CBO, DIT, and NOC values as seen in Table I. If the system has many inheritances, the difference will be large and the CCD will be larger than the CBO and CTA.

4.2.2. Measurement context

These measurements give the level of system understandability (i.e., the mental difficulties the participants experienced), the time to derive (reverse engineer) a class diagram from the source code, and the number of errors found in the diagram. Moreover, the level of understandability is composed of five levels: Level 1, Very Easy; Level 2, Easy; Level 3, Moderate; Level 4, Difficult; and Level 5, Very Difficult.

Here, the level of understandability reflects only the cognitive difficulty the participants feel. In general, people say that they cannot understand the problem if they feel it is difficult, and *vice versa*. The levels of cognitive difficulty or understandability are divided into five levels as stated earlier.

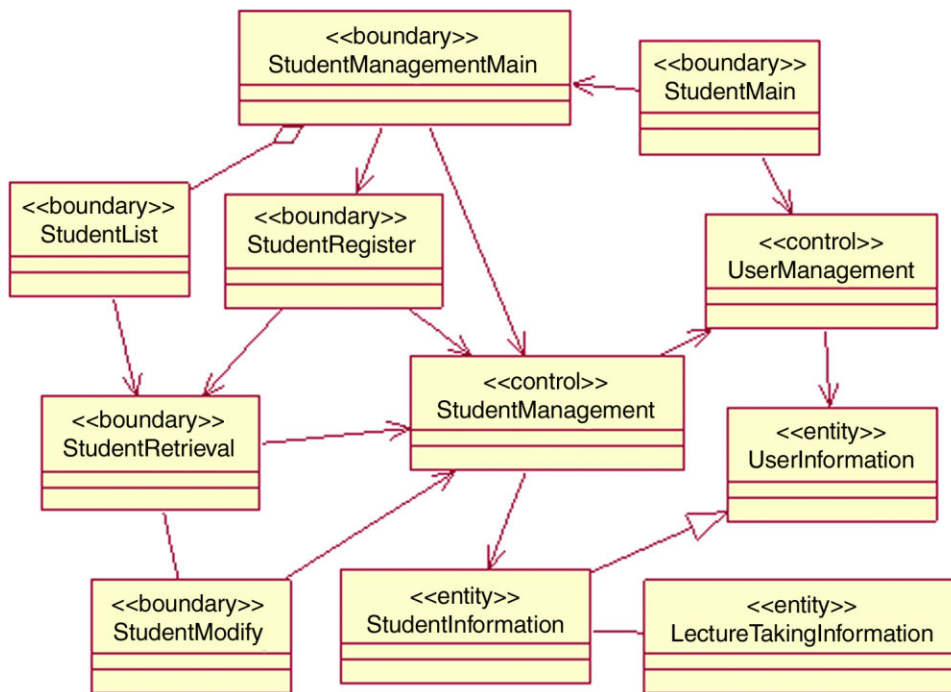


Figure 4. Class diagram for student management.

The reverse engineering time has intervals from 1 to 300 minutes. The participants have 10 minutes of rest after every 50 minutes of work, and they are prohibited from communicating with each other. The number of errors is counted from the diagrams the participants drew as compared to the class diagrams documented at the design phase of the system.

4.2.3. Experimental results

We allocated the six different cases (A–F) to the six groups (each group consisting of four undergraduates and one graduate) of participants and recorded the results shown in Table II. The values recorded in the row labeled undergraduate represents the average for the four undergraduate students. The row labeled average represents the team (group) score (sum of the above two values divided by two), which gives an inter-team comparative value. This average weights the results from graduate students as four times as important, which reflects the assumption that the graduate students performance would be more likely to be similar to experienced professionals. The average values of understandability, elapsed time in hours and the number of errors against an increment of CCD complexity from Table II are plotted in Figure 7. The average values of understandability and number

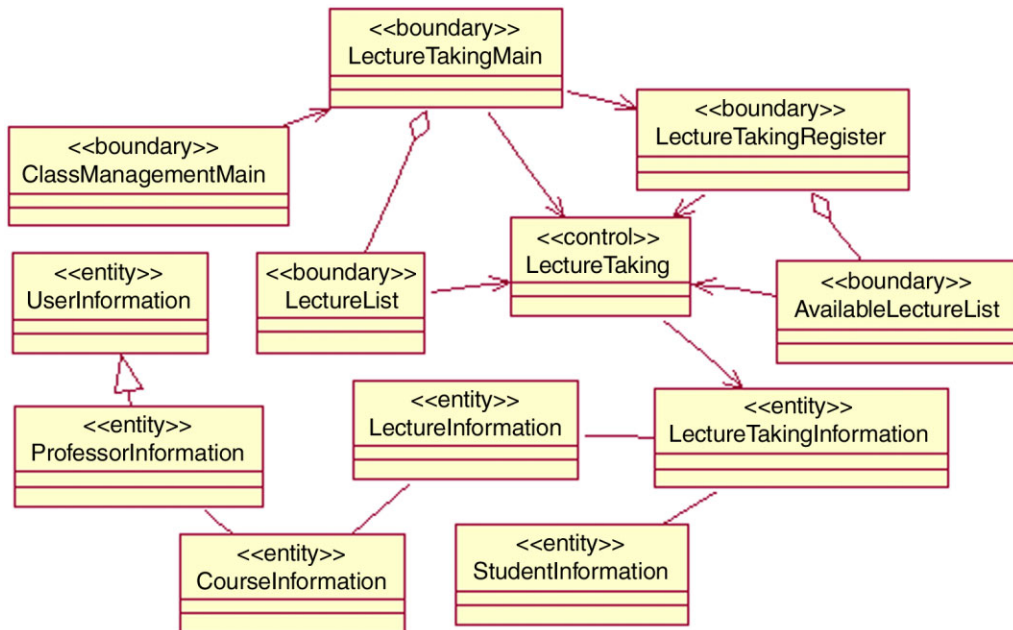


Figure 5. Class diagram for taking a lecture.

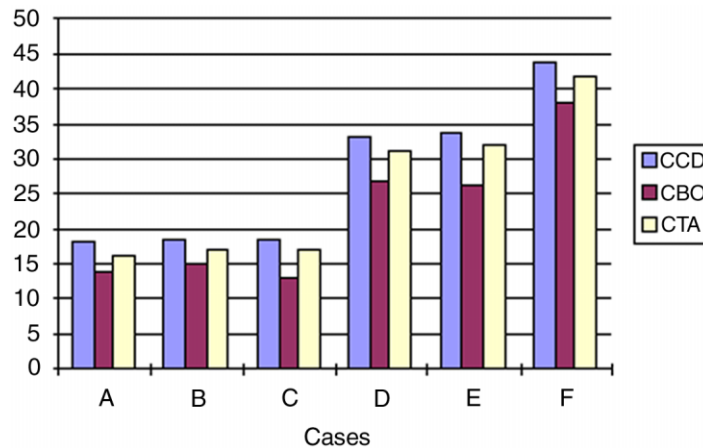


Figure 6. Plot of CCD, CBO and CTA for each case (A–F) as described in Table I.



Table I. Complexity as measured from the example class diagrams.

	Proposed metrics				C&K's metrics			Li's metrics		
	AR	AU	AGR	CCD	DIT	NOC	CBO	NAC	NDC	CTA
Case A	16	2	0	18	2	2	14	2	2	16
Case B	16	1.5	1	18.5	1	1	15	1	1	17
Case C	15	1.5	2	18.5	1	1	13	1	1	17
Case D	30	2	1	33	2	2	27	2	2	31
Case E	30	2	2	34	2	2	26	2	2	32
Case F	39	2	3	44	2	2	38	2	2	42

Table II. Experimental results from the six groups.

Cases	CCD	Participants	Understandability grading	Time (minutes)	Errors
A	18	Graduate	1	64	0
		Undergraduate	1	104	0.8
		Average	1	84 (1.4 hours)	0.4
B	18.5	Graduate	1	65	0
		Undergraduate	1.5	122	1.3
		Average	1.3	94 (1.57 hours)	0.7
C	18.5	Graduate	1	71	0
		Undergraduate	2.3	109	2.3
		Average	1.7	90 (1.5 hours)	1.1
D	33	Graduate	2	156	1
		Undergraduate	3.3	220	3.5
		Average	2.7	188 (3.13 hours)	2.3
E	34	Graduate	2	135	2
		Undergraduate	3.5	212	3.5
		Average	2.8	174 (2.9 hours)	2.8
F	44	Graduate	4	283	4
		Undergraduate	5	300	8.8
		Average	4.5	292 (4.87 hours)	6.4

of errors were normalized to make their values comparable to those of the elapsed time. In this way, we can show all three measures on the same graph with the same y-axis scale (0–350). Normalized here means that we manipulated the numbers using a linear adjustment to make them conform to the same scale. We found that all values for the level of understandability, the elapsed time, and the number of errors are sharply increased when the value of CCD passes over 30. If a system with a lot of inheritance were used in the experiment, the curve pattern would be sharper.

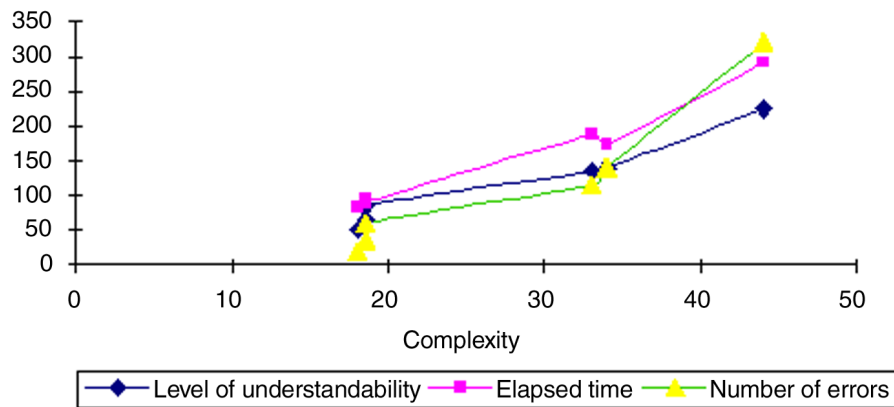


Figure 7. Plot of (y-axis) understandability, time, and errors against (x-axis) complexity as measured by CCD.

Factors influencing the complexity of a system are the complexity of the problem itself, the type of the system, the level of reliability, and so on. All factors are affected by the level of intelligence, experience, and problem comprehension from the standpoint of system developers. If the system were complex, it would be difficult to understand, therefore increasing the time needed to fully understand the system and increasing the amount of errors.

Participants who performed the experiment in which the complexity of the class diagram is about 30 indicated that the level of understandability is 'moderate'. Therefore, it is desirable that the complexity of a class diagram be below 30 when designing the structures of OO systems. However, the value of 30 would be a little different according to the judgment of each participant. If C&K's CBO and Li's CTA were plotted like the graph in Figure 7, the curve pattern would be similar. However, CBO and CTA are excluding the complexity of the inheritance relationship, so the overall complexity of the class diagram of a system would not be exact. This is because the CBO and CTA measure only the number of association relations between classes and do not measure inheritance, which makes the system difficult to understand. Therefore, CBO and CTA are not suitable for the measurement of complexity of the structures of systems that have large inheritance trees.

The process of designing an OO system involves defining the classes from the user's requirements specification, and then designing a class diagram that represents the static structure of the system by deriving the various relationships between classes. The class diagram largely influences the design of objects and the messages between them. Therefore, we must check the correctness and complexity of the diagram.

Correctness is determined by comparing the application to the user's requirements; however, complexity is also determined by analyzing the structures of the class diagram without considering the application. Among the metrics for measuring complexity, DIT and NOC proposed by C&K [3], NAC and NDC proposed by Li [5], and U and AU suggested by Sheldon *et al.* [6] only measure the inheritance relationship, and CTA proposed by Li [5] and COF suggested by Abreu [1] only measure



the association relationship. Most of such metrics measure the complexity of a part of a system and do not deal with the system as a whole, so it is difficult to determine the overall complexity of the system. Our study proposes and examines a structural complexity metric CCD (described in Section 4.1), in comparison to the other aforementioned metrics (i.e., C&K's DIT, NOC, and CBO metrics as well as Li's NAC, NDC, and CTA metrics), which can estimate the complexity of the system structures *as a whole* by combining all three aggregation, generalization, and association relationships.

5. CONCLUSION

Software developers often use UML class diagrams notation to design the static structures of a system. The literature is ripe with the definition/development and validation of OO complexity metrics. In this study we have performed a small case study to evaluate our new, holistic measure based on three different relationships that exist among classes in the class diagram: association, generalization, and aggregation. We have combined these three relations in such a way that enables a comprehensive measure of complexity. In theory, this measure is applicable among different class diagrams (including different domains or platforms/systems) to the extent that it is widely comparative and context free. This property does not preclude comparison within a specific class diagram (or family) and is therefore useful in evaluating a given class diagram's strength/weaknesses as a basis for refinement (i.e., in design or post deployment). We are not equating complexity with maintainability; rather, we are reporting empirical results that provide some measure of validity because complexity is often a good indicator for ease of maintenance. Consequently, to evaluate our CCD metric, we assessed system-level understandability by measuring the time needed to reverse engineer source code for a given class diagram and the number of errors produced while creating the diagram. The results show that a system with CCD complexity of 30 or more may significantly impede implementation and/or maintenance. The results as compared with other complexity metrics indicate that our CCD metric shows promise. Although not addressed in this study, we believe, because CCD combines and abstracts three different relationships, that CCD will work well for both large and small class diagrams (i.e., scalability). Further, CCD is both simple to compute and easy to understand. Acceptance by practitioners should be favorable.

Some weaknesses should be identified. A weakness exists in the theoretical validation because the metric is heuristically approached. Students participated in the experiment as the target measurement group; however, skilled software development programmers/engineers working in industry would be more ideally suited as the target. Also, the results of our study would have been more valid had we been able to evaluate actual deployed/fielded systems. Nonetheless, we believe that the pattern indicating an increase in the level of understandability (i.e., as demonstrated in the elapsed time and number of errors) would be comparable for the following reason. Applying objective measures (i.e., process time and product correctness) to the cognitive experience (process and results) of understanding depended primarily on the complexity of the representation form (subject matter) and not (so much) on the application.

Although it is difficult to know the absolute level of understandability, it seems reasonable to determine the relative and comparable level of understanding by objectively assessing the performance of the test subjects. This same line of reasoning is followed to assess how well students understand a subject matter by judging their performance on an exam, for example.



ACKNOWLEDGEMENTS

A contractor of the U.S. Government (USG) under DOE Contract DE-AC05-00OR22725 has co-authored this manuscript. The USG retains a non-exclusive, royalty-free license to publish or reproduce the published form of this contribution, or allow others to do so, for USG purposes.

The authors would like to thank the reviewers for their insights and constructive comments, including but not limited, to Kristopher Daley at Oak Ridge National Laboratory.

REFERENCES

1. Abreu F. The MOOD metrics set. *Proceedings of the ECOOP'95 Workshop on Metrics*, 1995.
2. Briand L, Daly J, Wust J. A unified framework for coupling measurement in object-oriented systems. *IEEE Transactions on Software Engineering* 1999; **25**(1):99–121.
3. Chidamber S, Kemerer C. A metrics suite for object-oriented design. *IEEE Transactions of Software Engineering* 1994; **20**(6):476–493.
4. Ferneley E. Coupling and control flow measures in practice. *The Journal of Systems and Software* 2000; **51**(2):99–109.
5. Li W. Another metric suite for object-oriented programming. *The Journal of Systems and Software* 1998; **44**(2):155–162.
6. Sheldon F, Jerath K, Chung H. Metrics for maintainability of class inheritance hierarchies. *Journal of Software Maintenance and Evolution: Research and Practice* 2002; **14**(3):147–160.
7. Basili V, Briand L, Melo W. A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on Software Engineering* 1996; **22**(10):751–761.
8. Briand L, Wust J, Daly J, Porter D. Exploring the relationships between design measures and software quality in object-oriented systems. *The Journal of Systems and Software* 2000; **65**(3):245–273.
9. Deligiannis I, Shepperd M, Roumeliotis M, Stamelos I. An empirical investigation of an object-oriented design heuristic for maintainability. *The Journal of Systems and Software* 2003; **65**(2):127–139.
10. Emam K, Melo W, Machado J. The prediction of faulty classes using object-oriented design metrics. *The Journal of Systems and Software* 2001; **56**(1):63–75.
11. Gursaran. Viewpoint representation validation: A case study on two metrics from the Chidamber and Kemerer suite. *The Journal of Systems and Software* 2001; **59**(1):83–97.
12. Harrison R, Counsell S, Nithi R. Experimental assessment of the effect of inheritance on the maintainability of object-oriented systems. *The Journal of Systems and Software* 2000; **52**(2–3):173–179.
13. Harrison R, Counsell S, Nithi R. An evaluation of the MOOD set of object-oriented software metrics. *IEEE Transactions on Software Engineering* 1998; **24**(6):491–496.
14. Henderson-Sellers B. *Object-Oriented Metrics: Measures of Complexity*. Prentice-Hall PTR: Upper Saddle River NJ, 1996; 252 pp.
15. Subramanian G, Corbin W. An empirical study of certain object-oriented software metrics. *The Journal of Systems and Software* 2001; **59**(1):57–63.
16. Roberts T. Metrics for object-oriented software development. *Addendum to the Proceedings on Object-oriented Programming Systems, Languages, and Applications (OOPSLA)*. ACM Press: New York NY, 1992; 97–100.
17. Chidamber SR, Kemerer CF. Towards a metrics suite for object oriented design. *Proceedings of the ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications (OOPSLA)*. ACM Press: New York NY, 1991; 197–211.
18. Holmboe CA. Cognitive framework for knowledge in informatics: The case of object-orientation. *Proceedings of the 4th Annual SIGCSE/SIGCUE ITICSE Conference on Innovation and Technology in Computer Science Education*. ACM Press: New York NY, 1999; 17–20.
19. Kim J, Lerch FJ. Towards a model of cognitive process in logical design: Comparing object-oriented and traditional functional decomposition software methodologies. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM Press: New York NY, 1992; 489–498.
20. Or-Bach R, Lavy I. Cognitive activities of abstraction in object orientation: An empirical study. *ACM SIGCSE Bulletin* 2004; **36**(2):82–86.
21. Robillard PN, d'Astous P, Detienne F, Visser W. Measuring cognitive activities in software engineering. *Proceedings 20th International Conference on Software Engineering*. IEEE Computer Society Press: Los Alamitos CA, 1998; 292–299.
22. Stacy W, MacMillan J. Cognitive bias in software engineering. *Communications of the ACM* 1995; **38**(6):57–63.
23. Chapin N. Coupling and strength, à la Harlan D. Mills. *Proceedings Science and Engineering for Software Development: A Recognition of Harlan D. Mills' Legacy*. IEEE Computer Society Press: Los Alamitos CA, 1999; 4–13.



24. Howell J, Miller P, Park HH, Sattler D, Schack T, Sperry E, Widhalm S, Palmquist M. Reliability and validity. *Writing Guides: Understanding Reliability and Validity*. Colorado State University, Department of English: Fort Collins CO, 2005. Available at: <http://writing.colostate.edu/guides/research/relival/> [April 2006].
25. Carmines EG, Zeller RA. *Reliability and Validity Assessment (Quantitative Applications in the Social Sciences*, No. 17). Sage Publications: Newbury Park CA, 1979; 72.
26. Chidamber SR, Darcy DP, Kermmerer CF. Managerial use of metrics for object-oriented software: An exploratory analysis. *IEEE Transactions of Software Engineering* 1998; **24**(8):629–639.

AUTHORS' BIOGRAPHIES



Frederick T. Sheldon has over 22 years of experience in the field of computer science and software engineering. He currently is a senior research staff scientist at Oak Ridge National Laboratory. Formerly, he was assistant professor at Washington State University and the University of Colorado at Colorado Springs, and research staff at DaimlerChrysler, Lockheed Martin Advanced Avionics, Raytheon, and NASA Langley/Ames. He received his PhD/MS at the University of Texas at Arlington in 1996/1989 (he also has two degrees from the University of Minnesota in Computer Science and Microbiology) and founded the Software Engineering for Secure and Dependable Systems Lab in 1999. He is a Senior Member of the IEEE and a member of ACM. He has published over 70 papers in various journals and international conferences (<http://www.csm.ornl.gov/~sheldon/pubs.html>). His research has been concerned with developing and validating/testing models, applications, methods, and supporting tools for the creation of safe, secure, and dependable software/systems.



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. He received his PhD at the Daegu Catholic University, Korea. He has worked as a Senior Researcher at the Computer Laboratory, Korea Institute of Science and Technology, Seoul, Korea, and also worked as a visiting scholar at the Washington State University, Pullman WA, U.S.A.