Measuring the Complexity of Class Diagrams in Reverse Engineering

The Complexity of Static Structures in Object-Oriented Systems are Studied and Compared by Analyzing UML Class Diagrams.

Frederick T. Sheldon and **Kristopher M. Daley**, Computational Science and Engineering, Oak Ridge National Laboratory[†],
Hong Chung, Department of Computer Engineering, Keimyung University, Daegu Korea[††]

Abstract

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 succeeded in evaluating only 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 and impact maintainability. The class diagram is composed of three kinds of relations: 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 and valid measure of complexity. Theoretically, 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. 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 strength/weaknesses. Further, we are not equating complexity with maintainability, rather, we are reporting empirical results that provide a small measure of validity against the backdrop of the complexity/maintainability question. Therefore, to evaluate our structural complexity metric, we measured the level of understandability of the system by measuring the time needed to reverse engineer the source code for a given class diagram including the number of errors produced while creating the diagram as one indicator of maintainability. The results as compared to other complexity metrics indicate our metric shows promise especially if proven to be scalable.

Keywords: perfective/corrective maintenance, object-oriented metrics, complexity metrics, and class diagram

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. The common thread (essential basis) 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 in the case for this work, the root concept deals with the complexity of object-oriented systems. Examples include the research of Chidamber and Kermerer [3] who proposed six object-oriented metrics (CK metrics) which are

[††]Grant Sponsor: This work was supported by grant No. RTI04-01-01 from the Regional Technology Innovation Program of the Ministry of Commerce, Industry and Energy (MOCIE), Korea

now often used to measure the complexity of object-oriented systems; Li's [5] proposed metrics that were aimed to overcome 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 object-oriented 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.

In this paper, we present an analysis of class diagrams used in the design of static structures using UML (Unified Modeling Language). UML is widely used and provides standardized object-oriented notations for specifying both the static and dynamic characteristics of OO Systems. This provides a solid basis for the assessment of overall system complexity. By measuring the complexity of a system during the design phase, we can reduce development cost as well as empower the modification and improvement of tasks during the implementation, maintenance, and evolutionary phases. Such measures can serve as a refactor to refine our architectural approaches. In evaluating the complexity of the structures of an OO system, we strived to measure the level of understandability of the system by measuring the time needed to reverse engineer the source code for a given class diagram, and then determining the number of errors that occurred during the process of creating the diagram. Our results indicate that the proposed metric has a considerably strong relationship to the complexity of the OO system.

In Section 2, the proposed metrics of Chidamber and Kemerer [3], Li [5], and Sheldon *et. al.* [6] are described and analyzed as the basic works of our research, and three types of relations between classes: association, generalization and aggregation are analyzed in detail in Section 3. In Section 4, we described the key functions of a university information system and present class diagrams of the structure of this system to suggest a new metric to measure the complexity of an OO system. We also performed an experiment to justify the validity of our metric by applying it to the system.

2. RELATED WORK

Chidamber and Kermerer [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. The complexity of a method can be 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 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 among the above six metrics 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 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 class inheritance hierarchy, but is conversely 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 whether they are visible outside the class or not. The local methods are all the public and private methods. The structural complexity can be LOC or CC. CTA (Coupling Through ADT) measures the coupling between classes 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 classes' services a class needs to provide 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.

Sheldon *et. al*. [6], suggested two kinds of metrics 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 DAG (Directed Acyclic Graph), 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 2 (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 Chidamber and Kemerer [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 relations. Further, we will suggest a unified metric considering all relationships for measuring the complexity of relations among the classes. In other words, a new metric of unifying DIT, NOC and CBO, or CTA, CTM, NAC and NDC will be proposed and validated.

3. ANALYSIS OF COMPLEXITY OF CLASS DIAGRAMS

The complexity of static structures represented by a class diagram is dependent on how the relationships between the classes are structured. Using the UML notations, 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.

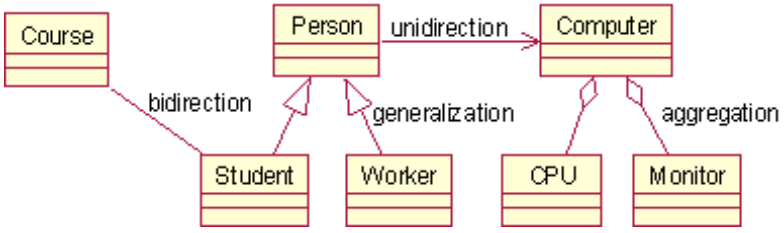

Figure 1. Simple class diagram used in the example descriptions.

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 here:

```
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, lets consider the bidirectional association between the class Person and the class Computer from Figure 1 given here:

```
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 in 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 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 example of the superclass Person and the subclasses Student and Worker of Figure 1 is shown here:

```
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 WhatCpmpanyl() {...}
   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 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 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 the 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 describing how the whole is related to its parts. Aggregation is known as a containment relationship. Often, it is ambiguous which relation we should use among aggregation and association, but

the key difference is that the aggregation includes all subclasses (i.e., the class Computer includes both classes CPU and Monitor) but the association does not. Another difference is that the association is symmetrical to the bidirectional relationship, while aggregation is not. The coding example of the aggregation among the class Computer, CPU, and Monitor in Figure 1 is shown below.

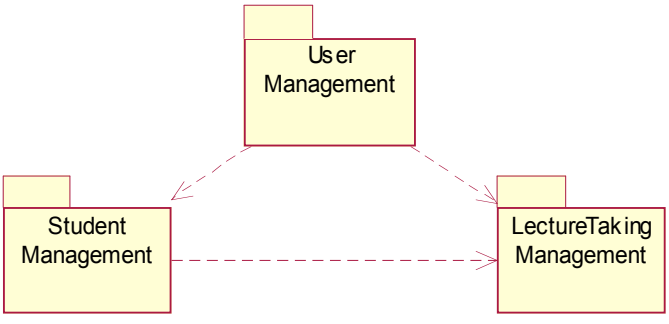

Figure 2. IS Package Diagram

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

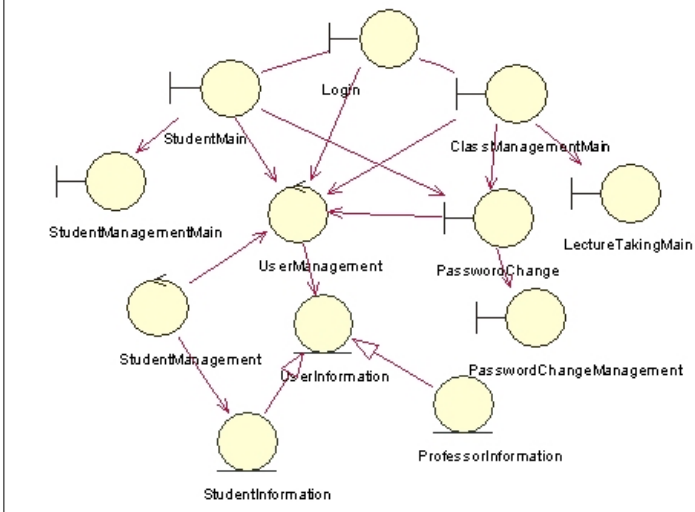

Figure 3. User management class diagram.

Since the class Computer references the classes CPU and Monitor, the class Computer must know both other classes. The aggregation relationship is considered to be a unidirectional association from the standpoint of the class structure, so it is easy to understand and the complexity is almost the same as the unidirectional association.

4. MEASURING THE COMPLEXITY OF OO SYSTEMS STATIC STRUCTURES

The complexity of the static structures of object-oriented (OO) systems is naturally measured by the number of connections between classes. Chidamber and Kermerer [3] proposed the metric (CBO) Coupling Between Objects 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] use 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. Clearly, the complexity of a bidirectional association is twice as difficult to understand compared to a unidirectional association because both fan-in and fan-out functionality are included simultaneously. Therefore, 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 Metric for the Complexity of a Class Diagram

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.

$$AR = (\text{No. of unidirectional relationships}) + 2*(\text{No. of bidirectional relationships}) \quad (1)$$

2) We use AU (Average Understandability) 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 unit value 1.

$$AU = (\Sigma(PRED(C_i)+1)) / n \quad (2)$$

Here, n is the number of classes in the inheritance tree, and $PRED(C_i)$ is the number of predecessor classes of the i*th* class C_i, i=1,…,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(s) when the inheritance tree is represented as a DAG (Directed Acyclic Graph). The average is computed because the generalization complexity of each class is different.

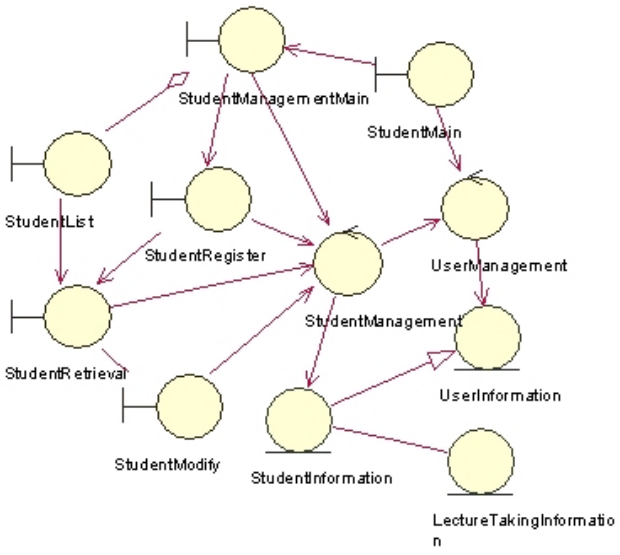

Figure 4. Student Management Class Diagram.

3) We assume the complexity of the aggregation relationship is equal to that of a unidirectional association. The aggregation class is a type of containment for all 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 (Aggregation Relationship) is defined as follows.

AGR = (No. of aggregation relationships) (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, the complexity of class diagram, CCD (Complexity of Class Diagram) can be expressed as a composition of the three expressions.

CCD=AR+AU+AGR (4)

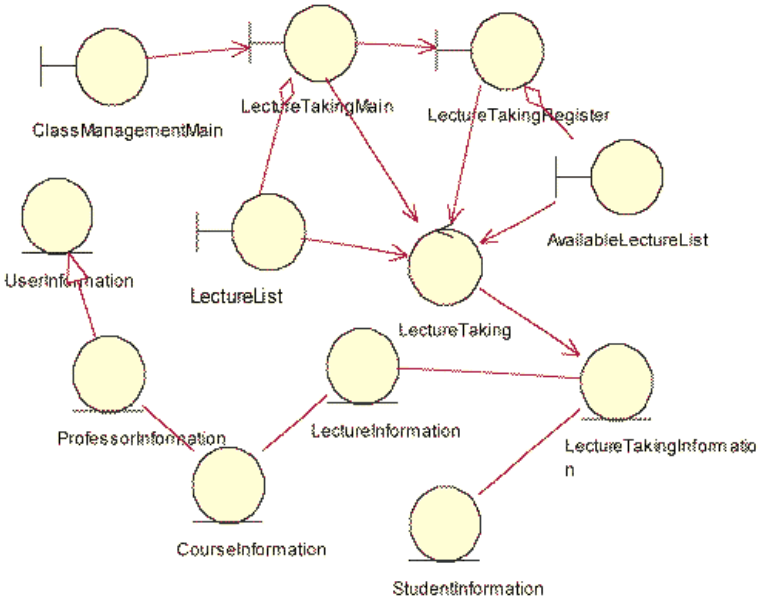

Figure 5. Lectures Taking Class Diagram

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

CCD={(No. of unidirectional relationships) + 2*(No. of bidirectional relationships)}

+ (Σ(PRED(Ci)+1))/n + (No. of aggregation relationships) (5)

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

CCD=1+2*1+(1+2+2)/3+2=6.67

The complexity of the class diagram in Figure 1 is considered very simple. In our judgment, a CCD below 30 should be considered relatively simple while one that is over 30 relatively complex. This judgment is based on 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.

Table 1. Complexity of class diagrams.

	Proposed metrics				**CK'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

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 6 subsystems as described in next section.

Table 2. Experimental results

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

Participants in the experiment included 24 undergraduate students who had taken the department's object-oriented systems course with a grade of B (or better), and 6 graduate students. Six groups consisting of 4 undergraduates and 1 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 form 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 3 ~ 5 junior programmers, so the results of our experiment, which included six distinct groups of one senior programmer and 4 junior programmers, would be considerably valid.

4.2.1 Structures from the System

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 Figure 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, CK's metrics and Li's metrics, from the class diagrams in Figure 3, 4 and 5 are in Table 1.

The values of CCD, CBO and CTA in Table 1 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 experiment has only one 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 of the OO paradigm. For example, in the system with 180 classes used in the 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.* [16], CBO is 22, but DIT and NOC are 2.

Similarly, our system has 38 of CBO value and 2 of DIT and NOC value as seen in Table 1. 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 Contents for Measurement

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 is only the cognitive difficulty the participants feel. In general, people say that they can't understand the problem if they feel it is difficult, and vice versa. The levels of the cognitive difficulty or understandability are generally divided into five levels as stated earlier.

The reverse engineering time has intervals from 1 minute to 300 minutes. The participants have 10 minutes of rest after every 50 minutes of work, and they are prohibited to communicate with each other. The number of errors is counted from the diagrams the participants draw, 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 of participants and recorded the results shown in Table 2. The values recorded in the row labeled undergraduate represents the average of 4 undergraduate students. Since the graduate students were highly skilled in programming and the undergraduates unskilled, the values recorded in the row labeled *average* is the average of the undergraduates and graduates (sum of the above two values divided by 2). The average values of understandability, elapsed time in hours and the number of errors against an increment of complexity from Table 2 are plotted in Figure 7. Multiplying their scale by 50 to better represent their complexity normalized the scale for the values of understandability and number of errors as compared to the elapsed time. We found that all values of 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 many inheritances is used in the experiment, the curve pattern would be sharper.

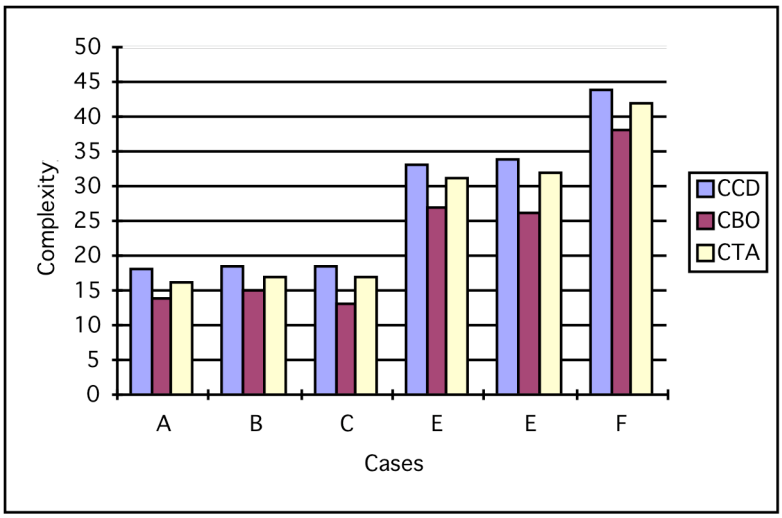

Figure 6. Plot of CCD, CBO and CTA of each case

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 have relations to the level of intelligence, experience and capability of understanding the problem in a standpoint of system developers. If the system is 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 object-oriented systems. But the value of 30 would be a little different according to the judgment of each participant. If CK's CBO and Li's CTA are plotted like the graph in Figure 7, the curve pattern would be similar. But 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 the 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 object-oriented system is by defining the classes from the user's requirements specification, and then designing a class diagram representing the static structure of the system by deriving the various relationships between classes. The class diagram largely influences

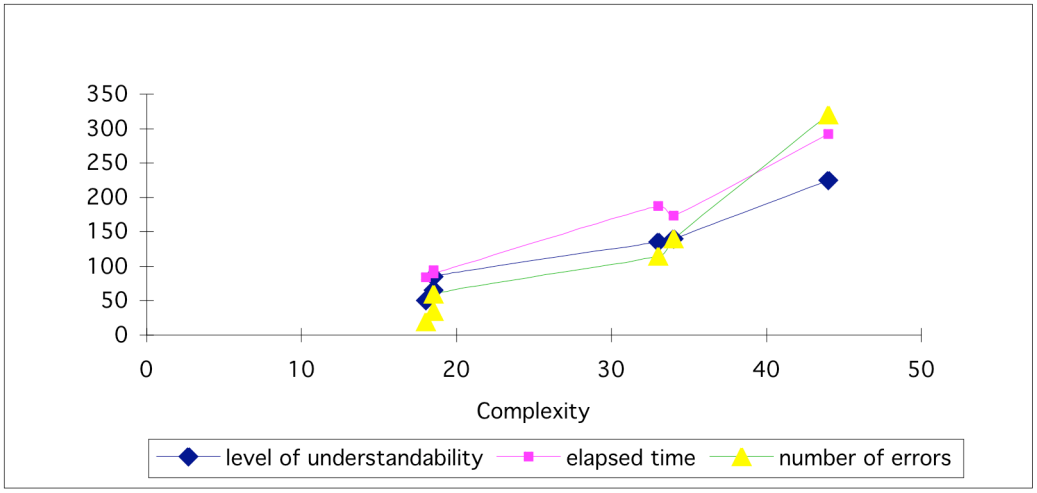

Figure 7. Plot of understandability, time and errors against complexity (y-axis).

the design of objects and messages between them. Therefore, we must check the correctness and complexity of the diagram.

The correctness is determined by comparing the application to the user's requirements; however, the complexity is determined by analyzing the structures of the class diagram without considering the application. Among the metrics for measuring complexity, DIT and NOC proposed by Chidamber and Kermerer [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 suggests a metric that can estimate the complexity of the system structures as a whole by way of synthesis of all relationships (aggregation, generalization and association).

5. CONCLUSION

OO system designers generally use UML class diagrams notations to design the static structures of a system. The literature is ripe with the definition/development and validation of OO complexity metrics. In contrast, we have defined/validated a new, more holistic measure based on three kinds of relationships among classes in the class diagram: association, generalization and aggregation, which make their overall structure difficult to understand. We have combined these three relations in such a way that enables a comprehensive and valid measure of complexity. Theoretically, 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. 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 strength/weaknesses as a basis for refinement (i.e., in design or post deployment). Further, we are not equating complexity with maintainability, rather, we are reporting empirical results that provide a measure of validity because we believe that complexity is a good predictor of maintainability. Therefore, to evaluate our metric, we measured the level of understandability of the system by measuring the time needed to reverse engineer source code for a given class diagram including the number of errors produced while creating the diagram as an indicator of understandability/maintainability for the given class diagram. The results show that a system with complexity of 30 or more may significantly impede implementation and the maintenance. The results as compared to other complexity metrics indicate our metric shows promise especially if proven to be scalable. Though not addressed in this study we believe, because the metric combines three relationships into one, thereby compressing/abstracting the relationship information, that the metric will work well for both large and small class diagrams. Further, the strength of the metric is derived from simplicity and ease of understanding. Consequently, 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 in the industrial field 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 the increased pattern in the level of understandability, 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. Though difficult to know the absolute level of understandability, it is reasonable to determine the relative and comparable level of understanding (e.g., grading exams on a curve).

6. REFERENCES

1. Abreu, F. *The MOOD Metrics Set*. in *ECOOP'95 Workshop on Metrics*. 1995.
2. Briand, L., J. Daly, and J. Wust, *A Unified Framework for Coupling Measurement in Object-Oriented Systems.* IEEE Transactions on Software Engineering, 1999. **25**(1): p. 99-121.
3. Chidamber, S. and C. Kermerer, *A Metrics Suite for Object-Oriented Design.* IEEE Transactions of Software Engineering, 1994. **20**(6): p. 476-493.
4. Ferneley, E., *Coupling and Control Flow Measures in Practice.* The Journal of Systems and Software, 2000. **51**(2): p. 99-109.
5. Li, W., *Another Metric Suite for Object-Oriented Programming.* The Journal of Systems and Software, 1998. **44**(2): p. 155-162.
6. Sheldon, F., K. Jerath, and H. Chung, *Metrics for Maintainability of Class Inheritance Hierarchies.* Journal of Software Maintenance and Evolution, 2002. **14**(3): p. 147-160.
7. Basili, V., L. Briand, and W. Melo, *A Validation of Object-Oriented Design Metrics as Quality Indicators.* IEEE Transactions on Software Engineering, 1996. **22**(10).
8. Briand, L., et al., *Exploring the Relationships between Design Measures and Software Quality in Object-Oriented Systems.* The Journal of Systems and Software, 2000. **65**(3): p. 245-273.
9. Deligiannis, I., et al., *An Empirical Investigation of an Object-Oriented Design Heuristic for Maintainability.* The Journal of Systems and Software, 2003. **65**(2): p. 127-139.
10. Emam, K., W. Melo, and J. Machado, *The Prediction of Faulty Classes using Object-Oriented Design Metrics.* The Journal of Systems and Software, 2001. **56**(1): p. 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): p. 83-97.
12. Harrison, R., S. Counsell, and R. Nithi, *Experimental Assessment of the Effect of Inheritance on the Maintainability of Object-Oriented Systems.* The Journal of Systems and Software, 2000. **52**(2-3): p. 173-179.
13. Harrison, R., S. Counsell, and R. Nithi, *An Evaluation of the MOOD Set of Object-Oriented Software Metrics.* IEEE Transactions on Software Engineering, 1998. **24**(6).
14. Henderson-Sellers, B., Object-Oriented Metrics: Measures of Complexity. 1996: Prentice Hall PTR.
15. Subramanian, G. and W. Corbin, *An Empirical Study of Certain Object-Oriented Software Metrics.* The Journal of Systems and Software, 2001. **59**(1): p. 57-63.
16. Chidamber, S.R., D.P. Darcy, and C.F. Kemerer, *Managerial Use of Metrics for Object-Oriented Software: An Exploratory Analysis.* IEEE Transactions of Software Engineering, 1998. **24**(8): p. 629-639.