ELSEVIER

The Journal of Systems and Software 44 (1998) 155–162

The Journal of Systems and Software

Another metric suite for object-oriented programming

Wei Li [1]

Computer Science Department, The University of Alabama in Huntsville, Huntsville, AL 35899, USA

Received 23 November 1997; received in revised form 8 January 1998; accepted 22 February 1998

Abstract

Chidamber and Kemerer (C&K) proposed six metrics for object-oriented programming. Discussions about the metrics were also reported. Recently, Kitchenham and her colleagues proposed a framework to validate software metrics. This paper evaluates C&K metrics by using Kitchenham's metric-evaluation framework and finds deficiencies in some of the C&K metrics. In order to remedy the deficiencies, this paper proposes another metric suite for object-oriented programming. The new metric suite, also six in number, includes Number of Ancestor Classes (NAC), Number of Local Methods (NLM), Class Method Complexity (CMC), Number of Descendent Classes (NDC), Coupling Through Abstract Data Type (CTA), and Coupling Through Message Passing (CTM).

Keywords: Object-oriented metrics; Metric validation; Metric-evaluation framework

1. Introduction

Since the proposal of the six object-oriented (OO) metrics by C&K (Chidamber and Kemerer, 1991), other researchers have made efforts to validate the metrics both theoretically and empirically. Li and Henry conducted an empirical study on the metrics using maintenance effort data collected from two commercial systems (Li and Henry, 1993). Basili et al. (1996) studied the metrics using software defects collected from student projects. C&K revised the original metrics and validated the metrics using measurement theory and empirical data (Chidamber and Kemerer, 1994). Discussions on possible ambiguities in some of the C&K metrics were also reported (Churcher and Shepperd, 1995; Chidamber and Kemerer, 1995; Hitz and Montazeri, 1996). The importance of validating software metrics was voiced by Pfleeger et al. (1997).

Suggestions of theoretical validation of software metrics include the use of measurement theory, Weyuker's metric-evaluation properties (Weyuker, 1988), and most recently, the framework by Kitchenham et al. (1995). This study uses Kitchenham's framework (thereafter referred to as the metric-evaluation framework) to evaluate C&K metrics.

In the remainder of this paper, the C&K metrics are evaluated against the metric-evaluation framework in Section 2. The deficiencies in C&K metrics are discussed. In Section 3, another suite of metrics for OO programming, which do not have the noted deficiencies, is proposed. Section 4 presents the empirical validations of the newly proposed metrics. The conclusion is in Section 5.

2. Evaluation of C&K metrics using the metric-evaluation framework

The metric-evaluation framework consists of five models: unit definition, instrumentation, attribute relationship, measurement protocol, and entity population (Kitchenham et al., 1995). In the unit definition model, a unit is defined for all measures including ratio, scale, nominal, and ordinal. An instrumentation model determines the method to capture a measure. When an attribute is composed of other attributes, the attribute relationship model defines the relationship among the attributes. A measurement protocol model is concerned with how to measure an attribute consistently on a specific entity. The entity population model sets the normal values for an attribute.

The metric-evaluation framework suggests four types of the unit definition models – reference to a standard, reference to a wider theory, reference to conversion from another unit, and reference to a model involving several attributes. Reference to a standard defines a metric unit

[1] Tel.: +1 205 890 6189; e-mail: wli@cs.uah.edu.

0164-1212/98/$ – see front matter
PII: S0164-1212(98)10052-3

based on a standard in an application domain. Reference to a wider theory determines the unit for a metric based on the way in which an attribute is observed on a particular entity. Reference to conversion from another unit sets a metric unit by converting from a known unit. Reference to a model defines the unit of a composite metric by combining the units of the individual metrics involved.

Instrumentation model is closely related to the unit definition model. The instrumentation model is used to take the measurements. There are two types of instrumentation models: the *direct representational* model and the *indirect theory-based* model. A direct representational model should be defined when a metric is measured directly from a software artifact such as a class or a function. A measure derived from a theory-based instrument is valid only if the underlying theory is valid.

There are two types of attribute relationship models: *definition* and *predicative*. The definition model defines a multidimensional attribute by expressing a relationship among attributes based directly on our understanding of the desired attributes. In a predictive model, the value of one or more attributes can be used to predict the value of another.

Measurement protocol model determines the measurement method so that the measure of a specific attribute on a specific entity is consistent and repeatable. The intention of the measurement protocol model is to make a measure independent of the measurer and the environment. The nature of the measurement protocol model makes it language dependent. Therefore, the definition of the model is left to empirical studies without further discussion in this paper.

Entity population model sets the normal values of a metric. Because defining normal values for the metrics is beyond the scope of this paper, it is also not discussed further in this paper.

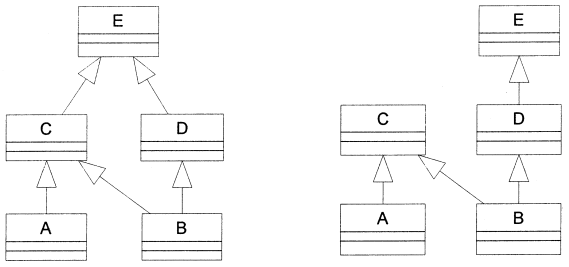
2.1. The depth of inheritance tree metric

The Depth of Inheritance Tree (DIT) metric is defined as "Depth of inheritance of the 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" (Chidamber and Kemerer, 1994). There are two ambiguous points in this definition.

First, this definition is ambiguous when multiple inheritance and multiple roots – classes that do not inherit from any other classes – are present at the same time. Consider the example in Fig. 1(a), the definition of the DIT metric applies well in calculating the DIT values for classes A and B, because the maximum lengths from nodes A and B to the root E can be determined precisely; thus, $DIT(A) = DIT(B) = 2$ in Fig. 1(a). In Fig. 1(b), however, the maximum length from node B becomes unclear. There are two roots in this design; the maximum length from node B to root C is one ($DIT(A) = 1$); the maximum length from node B to root E is two ($DIT(B) = 2$).

The second ambiguous factor lies in the conflicting goals stated in the definition, the theoretical basis, and the viewpoints for the DIT metric. The theoretical basis stated that "DIT is a measure of how many ancestor classes can potentially affect this class" (Chidamber and Kemerer, 1994). The viewpoints also stated that the DIT metric measures the number of classes that can potentially affect a class. Both the theoretical basis and the viewpoints seemed 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 apparently conflicts with the measurement attribute declared in the theoretical basis and the viewpoints. This ambiguous factor is only visible when multiple inheritance is present in an OO design, 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 visualized in Fig. 1(a).

In Fig. 1(a), according to the definition, classes A and B have the same maximum length from the root of the tree to the nodes respectively; thus $DIT(A) = DIT(B) = 2$. However, class B inherits from more classes than class A does. According to the theoretical basis and the viewpoints, classes A and B should



(a) Inheritance tree with single root. (b) Inheritance tree with multiple roots.

Fig. 1. Inheritance trees with single and multiple roots.

have different DIT values; thus DIT(A) = 2, and DIT(B) = 3.

No other models are proposed for the DIT metric because of these ambiguities. Instead, a new metric – the Number of Ancestor Classes (NAC) – is proposed as an alternative to the DIT metric. The NAC metric is defined in Section 3.

2.2. The number of children metric

The Number of Children (NOC) metric is defined as the "number of immediate subclasses subordinated to a class in the class hierarchy" (Chidamber and Kemerer, 1994). 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 nonimmediate. To remedy this insufficiency, a new metric – the Number of Descendent Classes (NDC) – is proposed as an alternative to the NOC metric; the NDC metric measures the NOC's stated theoretical basis and the viewpoints more closely than the NOC metric. The NDC metric is defined in Section 3.

Despite the inadequacy of the NOC metric from its stated viewpoints and theoretical basis, there is no difficulty in defining a unit for the metric. The unit of "class" is appropriated for the NOC metric because the metric measures the number of direct descendent classes of a class. This unit definition model is based on reference to a standard. The standard defines the immediate subclasses of a class in OO programming. In Fig. 1(a), NOC(A) = NOC(B) = 0 (class); NOC(C) = 2; NOC(D) = 1; NOC(E) = 2. In Fig. 1(b), NOC(A) = NOC(B) = 0; NOC(C) = 2; NOC(D) = 1; NOC(E) = 1.

There is no difficulty to define an instrumentation model for the NOC metric either. The instrumentation model for the NOC metric is clearly stated in its original definition (Chidamber and Kemerer, 1994).

2.3. The weighted method per class metric

The Weighted Method per Class (WMC) metric is defined as the sum of the complexity of a class' local methods. The WMC metric is intended to measure the combined complexity of a class' local methods. However, the definition, theoretical basis, viewpoints, and validation of the WMC metric leave the impression that the metric is used with two intentions: (1) the count of local methods, and (2) the sum of the internal complexity of all local methods. The number of local methods and the internal structural complexity of the local methods are two independent attributes of a class. If the WMC metric is used to measure the number of local methods of a class, the unit for WMC should be "method". If WMC is used to measure the total complexity exhibited by all local methods of a class, the WMC metric should have the same unit as the method's internal complexity, e.g. Lines of Code (LOC) or number of independent paths in a method, if McCabe's Cyclomatic Complexity (McCabe, 1976) is used to measure a method's complexity.

The WMC metric should measure the combined internal complexity of all local methods based on the definition. According to the theoretical basis, the WMC metric measures the number of local methods as per this description: "The number of methods is, therefore, a measure of class definition as well as being attributes of a class, since attributes correspond to properties" (Chidamber and Kemerer, 1994). The intention of using the WMC metric to measure the number of local methods is also clear in the viewpoints 2 and 3, the analytical evaluation, and the empirical data section (Chidamber and Kemerer, 1994).

The dual interpretations of the WMC metric create a dilemma for the proper usage of the metric because the metric measures two different attributes of a class; thus carrying two different units respectively. The two class attributes are independent. For example, a programmer who uses a class is mainly concerned with how many local methods are defined in the class, but not the internal complexity of each local method because of information hiding provided by OO programming. However, a programmer who maintains a class is interested in the internal complexity of all the local methods because the internal complexity of the methods affects the programmer's comprehension of the class as well as the effort required to maintain the class.

To demonstrate the two independent class attributes, consider the example in Fig. 2. The numbers of local methods defined in classes A and B are four and eight respectively; the size of class B's local interface, which is represented by the number of local methods, is twice as large as class A's. It is possible, however, that the internal complexity of class A's local interface is much more complicated than class B's. The difference between the two complexities of a class can hardly be distinguished by the WMC metric.

No models are proposed for the WMC metric because of the dual intentions of the metric. Instead, two metrics – Number of Local Methods (NLM) and Class Method Complexity (CMC) – are proposed to measure the two attributes that the WMC metric intends to capture. These two metrics are defined in Section 3.

2.4. The response for a class metric

The Response For a Class (RFC) metric measures the cardinality of "a set of methods that can potentially be executed in response to a message received by an object of that class" (Chidamber and Kemerer, 1994). No

```
class A {                          class B {
        int a;                             A anA;
public:                            public:
        void A();                          void B();
};                                 };
```

Fig. 2. Two class designs with different sizes and internal complexities of the local interfaces.

ambiguity or inadequacy is found in this metric. The unit for the RFC metric is "method". This unit is based on reference to a standard, which is the message passing concept in OO programming. The instrumentation model for the RFC metric is the means to calculate the RFC metric stated in (Chidamber and Kemerer, 1994).

2.5. The coupling between object classes metric

The Coupling Between Object Classes (CBO) metric is defined as "a count of the number of other classes to which it is coupled" (Chidamber and Kemerer, 1994). Although there is no difficulty in proposing the unit of "class" for the metric because the metric measures how many classes with which a class is coupled, it is difficult, however, to justify this unit. The unit definition model that most closely matches the unit of "class" for the CBO metric is reference to a standard. But there is no standard class coupling in OO programming. There are, however, different forms of class coupling such as inheritance, abstract data type, and message passing (Li, 1992). It is clearer to create a unit definition model for each of the three forms of class coupling than for the undefined class coupling in OO programming. The three forms of class coupling and the two metrics – Coupling Through Abstract Data Type (CTA), and Coupling Through Message Passing (CTM) – are defined in Section 3. No other models are suggested for the CBO metric.

2.6. The lack of cohesion in methods metric

The different definitions of the Lack of Cohesion in Methods (LCOM) metrics (Chidamber and Kemerer, 1991, 1994) were noticed and discussed (Hitz and Montazeri, 1996; Etzkorn et al., 1998). The original definition of LCOM metric measures the number of disjoint sets of a class' local methods as indicated by their access to class variables (Chidamber and Kemerer, 1991). The LCOM metric was later revised and a new definition was given (Chidamber and Kemerer, 1994). The revised LCOM metric is the count of pairs of instance variable sets (Chidamber and Kemerer, 1994). The discrepancy between the two LCOM definitions was noted by Hitz and Montazeri (1996). The definition of the revised LCOM indicates that LCOM is calculated as the number of pairs of instance variable sets in a class. However, the theoretical basis of the revised LCOM metric indicates that "The LCOM is a count of the number of method pairs whose similarity is 0 (i.e., $\sigma(\ )$ is a null set) minus the count of method pairs whose similarity is not zero" (Chidamber and Kemerer, 1994). This ambiguity concerning the LCOM metric makes it difficult, if not impossible, to define a unit for the revised LCOM metric. According to the definition, the LCOM metric should have a unit of "pair of instance variable set". According to the theoretical basis, the unit for the LCOM metric should be "pair of method set". Therefore, a unit definition and an instrumentation model are proposed only for the original LCOM metric which was interpreted by Li and Henry (1993), and later by Hitz and Montazeri (1996).

The unit for the LCOM metric is the "set of methods". This unit is proposed based on reference to a wider theory, which is Set Theory in Discrete Mathematics. The instrumentation model is the means to calculate the LCOM metric suggested by Li and Henry (1993) and later by Hitz and Montazeri (1996).

3. Another metric suite for OO programming

The problems associated with some of the C&K metrics were discovered during the course of defining the unit definition model for the metrics. An alternative suite of OO metrics that does not have the problems is proposed in this section. Each metric in the newly proposed suite measures a particular attribute of a class; the attribute is related to a specific concept in OO programming. In the definition of the newly proposed metrics, the metric-definition style which was used by Chidamber and Kemerer (1994) is used in addition to unit definition and instrumentation models (Kitchenham et al., 1995). Attribute relation model is not mentioned for any of the metrics because all the suggested metrics are primitive, not composite, metrics; therefore the attribute relation model does not apply to these metrics. No measurement protocol model is provided

for the metrics because the model should be left to empirical studies as suggested by Kitchenham and her colleagues (Kitchenham et al., 1995). No entity population model is given because that defining normal values for the metrics is beyond the scope of this paper. The measurement target of all the proposed metrics is a class in OO programming.

3.1. The number of ancestor class metric

The attribute of a class that the DIT metric intends to capture is the number of classes that have potential influence on the class because of the inheritance relations. This attribute is best captured by the number of ancestor classes of a class, regardless of the number of roots or whether multiple inheritance is present. The Number of Ancestor Classes (NAC) metric is proposed, as an alternative to the DIT metric, to measure this attribute of a class. The unit for the NAC metric is "class" because the attribute that the NAC metric captures is the number of other classes' environments from which the class inherits. This unit is defined with reference to a standard. The standard is class inheritance relation in OO programming. This unit definition model gives two classes different NAC values if the two classes are at the same level in an inheritance tree but inherit from different number of classes respectively.

Metric 1: Number of Ancestor Classes (NAC)

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

Theoretical basis: same as the DIT metric (Chidamber and Kemerer, 1994).

Viewpoints: same as the DIT metric (Chidamber and Kemerer, 1994).

Unit definition model: The unit for the NAC is "class". This unit is defined based on reference to a standard. The standard is the number of ancestor classes of a class in the inheritance tree(s) in OO programming.

Instrumentation model: The inheritance hierarchy of an OO design can be represented as a directed graph. The NAC metric is the number of nodes (classes) that can be reached from a particular node (the class) in the directed graph.

Based on the unit definition and the instrumentation models, there should be no ambiguity on the NAC values for class A and B in both Fig. 1(a) and (b). In Fig. 1(a), class A inherits from classes C and E, therefore, yielding NAC(A) = 2 (classes); class B inherits from three classes (C, D, and E), thus, yielding NAC(B) = 3. In Fig. 1(b), class A inherits from one class (C), thus, yielding NAC(A) = 1; class B inherits from three classes (C, D, and E), therefore, yielding NAC(B) = 3.

3.2. The number of descendent classes metric

The Number of Descendent Classes (NDC) metric is proposed as an alternative to the NOC metric. The attribute of a class that the NOC metric captures is the number of classes that may potentially be influenced by the class because of inheritance relations. This influence comes in two forms: data attributes and methods that are inheritable from the class by its subclasses. A class influences all its subclasses, not just the immediate ones. The NDC metric captures the class attribute stated for the NOC metric better than NOC.

Metric 2: Number of Descendent Classes (NDC)

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

Theoretical basis: same as the NOC metric in (Chidamber and Kemerer, 1994).

Viewpoints: same as the NOC metric in (Chidamber and Kemerer, 1994).

Unit definition model: The unit for the NDC is "class". This unit is defined based on reference to a standard. The standard is the number of descendent classes of a class in the inheritance tree(s) in OO programming.

Instrumentation model: The inheritance hierarchy of an OO design can be represented as a directed graph. The NDC metric is the number of nodes (classes) that the node (class) can reach through the directed graph.

3.3. The number of local methods and the class method complexity metrics

Two metrics – Number of Local Methods (NLM) and Class Method Complexity (CMC) – are proposed to measure the two attributes of a class that the WMC metric intends to capture.

Metric 3: Number of Local Methods (NLM)

Definition: The NLM metric is the number of the local methods defined in a class which are accessible outside the class (e.g. public methods in C++).

Theoretical basis: The attribute of a class that the NLM metric captures is the local interface of a class. This attribute is important for the usage of the class in an OO design because it indicates the size of a class' local interface through which other classes can use the class.

Viewpoints:

1. The NLM metric is directly linked to a programmer's comprehension effort when a class is reused in an OO design. The more local methods a class has, the more effort is required to comprehend the class' behavior.
2. The larger the local interface of a class, the more effort is needed to design, implement, test, and maintain the class.

3. The larger the local interface of a class, the more influence the class has on its descendent classes.

Unit definition model: The unit for the NLM metric is "method". This unit is based on reference to a standard. The standard is the local interface of a class in OO programming. The local interface of a class provides a means for other classes to access its encapsulated data (Li, 1992).

Instrumentation model: The total number of local methods that can be directly invoked from outside the class is the value of the NLM metric.

Metric 4: Class Method Complexity (CMC)

Definition: The CMC metric is the summation of the internal structural complexity of all local methods, regardless whether they are visible outside the class or not (e.g. all the public and private methods in C++). This definition is essentially the same as the first definition of the WMC metric in (Chidamber and Kemerer, 1991). However, the CMC metric's theoretical basis and viewpoints are significantly different from the WMC metric.

Theoretical basis: The CMC metric captures the complexity of information hiding in the local methods of a class. This attribute is important for the creation of the class in an OO design because the complexity of the information hiding gives an indication of the amount of effort needed to design, implement, test, and maintain the class.

Viewpoints:

1. The CMC metric is directly linked to the effort needed to design, implement, test, and maintain a class. The more complex a class' methods are, the more effort is needed to design, implement, test, and maintain the methods.
2. The more complex a class' methods are, as measured by the internal complexity of the methods, the more effort is needed to comprehend the realization of information hiding in a class.

Unit definition model: The unit for the CMC metric is defined by reference to conversion from another unit. The conversion depends on the unit used to measure the structural complexity of a method. For example, if LOC metric is used to measure the complexity of a method, the unit for the WMC metric should be converted from the LOC metric unit. If McCabe's Cyclomatic Complexity (CC) (McCabe, 1976) is used to measure the complexity of a method, the CMC metric unit should be the same as the unit for the CC metric, which is the number of independent paths within a function.

Instrumentation model: The sum of the internal complexity of all local methods, regardless of whether the methods are visible outside the class or not, is the instrumentation model.

The NLM and CMC metrics are fundamentally different because they capture two independent attributes of a class. There is, however, a commonality in the viewpoints of the two metrics – they both affect the effort required to design, implement, test, and maintain a class. This commonality makes sense because the effort needed to design, implement, test, and maintain a class depends on both the number of local methods and the internal structural complexity of all the local methods.

3.4. The coupling through abstract data type and the coupling through message passing metrics

A class can be coupled with other classes in three different forms – inheritance, abstract data type, and message passing (Li, 1992).

When a class inherits from another class, directly or indirectly, the two classes are coupled. This type of class coupling breaks the encapsulation that OO programming provides. No metric is proposed to measure this class attribute because the NAC and NDC metrics, or some composite metrics made of the NAC and the NDC metrics, can capture this attribute well.

Two classes are coupled when one class uses the other class as an abstract data type. Consider the example in Fig. 3. Class B is coupled with class A through the use of abstract data type because class B uses class A in its data-attribute declaration. For this form of class coupling, I propose the Coupling Through Abstract Data Type (CTA) metric.

Two classes can be coupled because one class sends a message to an object of another class, without involving the two classes through inheritance or abstract data

```
class A {                          class B {
        int a;                             A anA;
public:                            public:
        void A();                          void B();
};                                 };
```

Fig. 3. An example of coupling through abstract data type.

types. Consider the example in Fig. 4. Both classes A and B are in the same OO design, and they are not related through inheritance or abstract data type as a class attribute. However, class A is coupled with class B because one of A's methods sends a message to B's object. For a class, there are two aspects to message-passing coupling. One aspect is the scale to which the class can receive messages from other classes. The other aspect is the scope to which the class sends out messages to other classes. Some combinations of the NLM and the NAC metrics can measure the scope to which a class receives messages. Therefore, the discussion of message-passing coupling is limited to the scope to which the class sends out messages to other classes. The Coupling Through Message Passing (CTM) metric is appropriate to measure this type of class coupling.

Metric 5: Coupling Through Abstract Data Type (CTA)

Definition: The CTA metric is the total number of classes that are used as abstract data types in the data-attribute declaration of a class.

Theoretical basis: The CTA metric relates to the notion of class coupling through the use of abstract data types. This metric gives the scope of how many other classes' services a class needs in order to provide its own service to others.

Viewpoints:

1. A software engineer needs to spend more time in understanding the interfaces of the used classes in order to create the design for a high CTA class than a low one.
2. For a test engineer, more effort is needed to design test cases and perform testing for a high CTA class than a low one because that the behaviors of the used classes also need to be tested.
3. For a maintenance engineer, it takes more time to understand a high CTA class than a low one because a high CTA class uses more classes whose behaviors may complicate the class.

Unit definition model: The unit for the CTA metric is "class". This unit is defined based on reference to a standard, which is the use of abstract data type in OO programming.

Instrumentation model: A class contains two parts: the data attributes and the operation attributes. The type of each data attribute is defined with the name of the attribute. An analysis of each data type in a class' data attribute declaration should yield the number of different classes used as abstract data types in the class.

Metric 6: Coupling Through Message Passing (CTM)

Definition: The CTM metric measures the number of different messages sent out from a class to other classes excluding the messages sent to the objects created as local objects in the local methods of the class.

Theoretical basis: The CTM metric relates to the notion of message passing in OO programming. The metric gives an indication of how many methods (services) of other classes are needed to fulfill the class' own functionality.

Viewpoints:

1. A class designer needs to spend more effort in understanding the services provided by other classes in a high CTM class than in a low CTM class because the outgoing messages are directly related to the services other classes provide.
2. A test engineer needs to spend more effort and design more test cases for a high CTM class than for a low CTM class because a high CTM value means more other classes' methods are involved in the logical paths of the class.
3. For a maintenance engineer, the higher the CTM metric value, the more specific methods in other classes the engineer needs to understand in order to diagnose and fix problems, or to perform other types of maintenance.

Unit definition model: The unit for the CTM metric is "message". This unit is defined based on reference to a standard, which is the message-passing concept in OO programming.

Instrumentation model: There are two components in a message: (1) the object, and (2) the method. For a message to be counted, the message must be different

```
class A {                                   class B {
        int a;                                      A anA;
public:                                     public:
        void A();                                   void B();
        void A1 (B* b) { b->B1();};                 void B1 ();
};                                          };
```

Fig. 4. An example of coupling through message passing.

from the previously counted messages in either the object or the method. The messages sent to an object that is created, as a local object in a local method, is not counted in the CTM metric. Only the objects that are created as class data attributes or passed as parameters to local methods are counted in the CTM metric.

There are significant differences between the CTA and CTM metrics, although their viewpoints may resemble similarity. Not only do they carry different units, but they also measure two complete different attributes of a class. The high value of CTA metric in a class does not necessarily mean a high value of the CTM metric; and vice versa.

4. Empirical validation of the newly proposed metrics

It is important that software metrics be validated both theoretically and empirically. The evaluation of the newly proposed metrics using the metric-evaluation framework, suggested by Kitchenham and her colleagues, provides one theoretical. All but the NDC metrics have been empirically validated in Li and Henry's study (Li and Henry, 1993), although most of the metrics were used under different names. The NAC metric was validated indirectly through the validation of the DIT metric because there were no multiple inheritance and multiple roots present in the two systems used in Li and Henry's study; therefore, the DIT and NAC metrics had the same value for each class. The NLM, CMC, CTA, and CTM metrics were validated directly in the same study under different names; the metrics in Li and Henry's study which correspond to the newly proposed metrics were NOM, WMC, DAC, and MPC respectively.

5. Conclusion

The metrics for object-oriented programming defined by Chidamber and Kemerer were evaluated using the metric-evaluation framework proposed by Kitchenham and her colleagues. Some deficiencies of the metrics were discovered in the evaluation process. A new suite of OO metrics that does not have the noted deficiencies was proposed. The unit definition and instrumentation models were defined for each new metric. The evaluation of the newly proposed metrics using the metric-evaluation framework provides one theoretical validation. The empirical validation of all, but the NDC, metrics is discussed. The new suite of metrics is an attempt toward a more rigorous approach in OO metric study. The newly proposed suite of OO metrics is intended to complement and strengthen the metrics proposed by Chidamber and Kemerer.

Acknowledgements

I would like to thank the anonymous reviewers. Special thanks go to Dr. Letha Etzkorn and Dr. Jagdish Bansiya for their reviews and comments on an earlier version of this paper.

References

Basili, V.L., Briand, L., Melo, W.L., 1996. A validation of object-oriented metrics as quality indicators. IEEE Trans. Software Eng. 10, 751–761.

Chidamber, S.R., Kemerer, C.F., 1991. Towards a metrics suite for object oriented design. In: Proceedings of The Sixth Object-Oriented Programming Systems, Languages, and Applications, pp. 97–211.

Chidamber, S.R., Kemerer, C.F., 1994. A metrics suite for object oriented design. IEEE Trans. Software Eng. 6, 476–493.

Chidamber, S.R., Kemerer, C.F., 1995. Reply: Comments on a metrics suite for object oriented design. IEEE Trans. Software Eng. 3, 265.

Churcher, N.I., Shepperd, M.J., 1995. Comments on a metrics suite for object oriented design. IEEE Trans. Software Eng. 3, 263–265.

Etzkorn, L., Davis, C., Li, W., 1998. A practical look at the lack of cohesion in methods metrics. Object Oriented Programming 5, 42–46.

Hitz, M., Montazeri, B., 1996. Chidamber and Kemerer's metrics suite: A measurement theory perspective. IEEE Trans. Software Eng. 4, 267–271.

Kitchenham, B., Pfleeger, S.L., Fenton, N., 1995. Towards a framework for software measurement validation. IEEE Trans. Software Eng. 12, 929–944.

Li, W., 1992. Applying Software Maintenance Metrics in the Object-Oriented Software Development Life Cycle. Ph.D. Dissertation, Virginia Polytechnic Institute and State University, Blacksburg, Virginia.

Li, W., Henry, S., 1993. Object-oriented metrics which predict maintainability. J. Systems Software 2, 111–122.

McCabe, T., 1976. A complexity measure. IEEE Trans. Software Eng. 2, 308–320.

Pfleeger, S.L., Jeffery, R., Curtis, B., Kitchenham, B., 1997. Status report on software measurement. IEEE Software 2, 33–43.

Weyuker, E., 1988. Evaluating software complexity measures. IEEE Trans. Software Eng. 9, 1357–1365.

Wei Li received a B.S. in Computer Science from Beijing University and a Ph.D. in Computer Science from Virginia Tech. His areas of interest are object-oriented programming, software metrics, and reusable components. Dr. Li is a member of the ACM and IEEE Computer Society.