Developing Measures of Class Cohesion for Object-Oriented Software*

Linda Ott
Michigan Technological University

James M. Bieman
Colorado State University

Byung-Kyoo Kang
Colorado State University

Bindu Mehra
Michigan Technological University

Published in
Proc. Annual Oregon Workshop on Software Metrics (AOWSM'95)
June 1995

Abstract

Cohesion refers to the relatedness of module components and is a well-understood concept in the procedural paradigm. In the object-oriented paradigm, a concept of *class cohesion* appears to be necessary. In this paper, we compare two different approaches to measuring class cohesion.

1 Introduction

Seldom have new design techniques been evaluated in a rigorous and quantitative fashion. The object-oriented paradigm is no exception. Few quantitative studies have been conducted to verify claims concerning the reliability, maintainability, and reusability of software developed using object-oriented techniques. Such studies require appropriate measurement tools.

As with procedural software, we would like to be able to relate object-oriented structural quality to critical reliability, maintainability, and reusability process attributes. We need appropriate measures of object-oriented structure to begin to relate structure to process. The development of measures of structure appropriate to object-oriented systems has just begun. One example is the work of Chidamber and Kemerer who developed a suite of size and structure measures [5]. Definitions of this suite were further refined by Churcher and Shepperd [6], who also developed a conceptual framework for object-oriented metrics based on an entity-relationship model [7].

*J. Bieman and B-K. Kang are partially supported by NASA Langley Research Center grant NAG1-1461.

L. Ott is with the Department of Computer Science, Michigan Technological University, Houghton, MI 49931, Email: linda@cs.mtu.edu, (906) 487-2187.

J. Bieman is with the Department of Computer Science, Colorado State University, Fort Collins, CO 80523. Email: bieman@cs.colostate.edu, (303)491-7096, Fax: (303) 491-2466.

B-K. Kang is with the Department of Computer Science, Colorado State University, Fort Collins, CO 80523. Email: kang@cs.colostate.edu.

Bindu Mehra is with the Department of Computer Science, Michigan Technological University, Houghton, MI 49931, Email: bindu@cs.mtu.edu

Cohesion and coupling are two structural attributes whose importance is well-recognized in the software engineering community. Here we focus on cohesion. Cohesion refers to the "relatedness" of module components. In a procedural paradigm, a highly cohesive component is one with one basic function. It should be difficult to split a cohesive component. Cohesion can be classified using an ordinal scale that ranges from the least desirable category—*coincidental cohesion*—to the most desirable—*functional cohesion* [11].

In the past, our joint work has focused on the rigorous development of functional cohesion measures based on program slices [2]. Functional cohesion is an appropriate measure for procedural code where basic software units are procedures and functions. In object-oriented software, the basic design units are classes, which are collections of instance variables and methods. Functional cohesion cannot be applied directly to classes. Fenton suggests that what we are interested in here is a different form of cohesion, namely *data cohesion*, rather than *functional cohesion* [9].

We really need to develop a notion of class cohesion, which is an indicator of the degree to which the components of a class belong together. We have been working on developing class cohesion measures both separately and together. By examining the ideas of class cohesion from several different viewpoints and through independently developed measures, we hope to come to a better understanding of what is meant by cohesion in the object-oriented paradigm and the role cohesion plays in the development of quality object-oriented systems.

Ott and Mehra's approach is a direct extension of our work on functional cohesion. Bieman and Kang's approach evolved from Chidamber and Kemerer's measure of the lack of cohesion (LCOM) between methods [5]. In both approaches, we treat the class as the basic unit and the instance variables as the "glue" that connects the methods in a class. The difference in the approaches is in how we measure the "glue".

2 Ott and Mehra's approach.

In [2], functional cohesion measures are presented based on a slice abstraction model of programs. The underlying assumption is that in a procedural paradigm the intent of a module is to perform a function which is, in general, manifested by the computation of one or more outputs. Modules are assumed to be cohesive if the computations involved are related and therefore belong together. In order to measure the relatedness of the computations, data slices are computed for each output of a module where a data slice represents the data tokens in the module which have an effect on that output or are affected by that output. Data tokens which belong to more than one slice are considered "glue tokens" or indicators that the computations are related. The number of "glue tokens" and the total number of data tokens are the basis for the measures of cohesion.

2.1 Class Slicing

In [10], Linda Ott and Bindu Mehra modify the above approach in an attempt to measure the cohesion of classes in an object-oriented paradigm. Again, we assume that cohesion is an indication that the elements belong together, however, here we are referring to the elements of a class. In the object-oriented paradigm rather than assuming that the intent of a module is to perform some function, we assume that the intent of a class is to model some object which is represented by its behavior as reflected through its methods and by its state information maintained in the class private data or instance variables. Thus, we chose to develop a slice-based model where the slicing is based on the class instance variables. Slices are obtained for each of the class methods and

then concatenated together to form the slice model. Again, we measure the cohesion based on the number of data tokens that appear in more than one slice and thus "glue" the module together.

Figure 1 is an example of a data slice profile for class *list*. For each statement included in the data slice, the number of data tokens included in the slice from that particular statement is given. This is done for every method of the class as the slice profile for a class is the concatenation of the slice profiles of each method of the class.

We define a *class slice abstraction* of a class C, *CSA(C)*, to be the set of concatenated slices, one for each instance variable, formed by concatenating the data tokens obtained from the method data slices for that instance variable. Thus, the slice abstraction for class *list* of Figure 1 is:

$$CSA(list) =$$

$$\{head_1 \cdot NULL_1 \cdot someitem_1 \cdot pt_1 \cdot pt_2 \cdot item_1 \cdot someitem_2 \cdot pt_3 \cdot 0_1 \cdot is\_empty_1 \cdot head_2 \cdot pt_4 \cdot tail_2 \cdot next_1 \cdot pt_5 \cdot pt_6 \cdot next_2 \cdot NULL_3 \cdot tail_3 \cdot pt_7 \cdot pt_8 \cdot temp_1 \cdot pt_9 \cdot head_3 \cdot pt_{10} \cdot temp_2 \cdot pt_{11} \cdot pt_{12} \cdot pt_{13} \cdot next_3 \cdot temp_3 \cdot head_4 \cdot tail_4 \cdot NULL_4 \cdot head_5 \cdot NULL_5 \cdot true_1 \cdot false_1,$$

$$tail_1 \cdot tail_2 \cdot NULL_2 \cdot someitem_1 \cdot pt_1 \cdot pt_2 \cdot item_1 \cdot someitem_2 \cdot pt_3 \cdot 0_1 \cdot is\_empty_1 \cdot head_2 \cdot pt_4 \cdot tail_3 \cdot next_1 \cdot pt_5 \cdot pt_6 \cdot next_2 \cdot NULL_3 \cdot tail_3 \cdot pt_7 \cdot pt_8 \cdot temp_1 \cdot pt_9 \cdot head_3 \cdot pt_{10} \cdot temp_2 \cdot pt_{11} \cdot pt_{12} \cdot pt_{13} \cdot next_3 \cdot temp_3 \cdot head_4 \cdot tail_4 \cdot NULL_4\}$$

$head_1$, $NULL_1$ are the slice abstractions of method list() for slice data token *head*, and the other data tokens similarly are data slices of other methods for the slice data token *head*; they are combined to form the data slice for *head* for the class *list*. Similarly, a data slice for the slice data token *tail* can be obtained for the class *list*.

2.2 Data Cohesion Measures

The data cohesion measures are described in terms of slice abstractions, data tokens, and glue and super-glue tokens as applied to the object-oriented paradigm. The basic definition for a data token and glue and super-glue tokens are similar to those used in the procedural paradigm, that is, glue tokens are those data tokens that are elements of more than one data slice and super-glue tokens are those glue tokens that are elements of all the data slices. The super-glue tokens for the class, denoted as, SG(CSA(C)), is a union of the super-glue tokens of each of the methods of the class. Similarly, the set of glue tokens for the class, denoted as, G(CSA(C)), are a union of the glue tokens of each of the member methods of the class. *tokens(C)* is a set of all data tokens of a class C. The following measures are based on the measures in [2] and have been modified to use the concepts presented here for the object-oriented paradigm.

Strong Data Cohesion is a measure based on the number of data tokens included in all the data slices for a class, i.e. a count of the number of super-glue tokens in the class C. Thus, a class with no super-glue tokens will have zero strong data cohesion.

$$SDC(C) = \frac{|SG(CSA(C))|}{|tokens(C)|}$$

Weak Data Cohesion measures the amount of data cohesion in a class based on the glue tokens. Glue tokens, unlike the super-glue tokens, do not necessarily bind together all the data slices, hence are indicative of a weaker type of cohesion. A class with no glue tokens will have no weak data cohesion.

$$WDC(C) = \frac{|G(CSA(C))|}{|tokens(C)|}$$

head	tail	Class List
		`list::list(){`
2		`   head = NULL;`
	2	`   tail = NULL;`
		`}`
		`list::~list()`
		`{`
		`    remove();`
		`}`
1	1	`void list::append(Type someitem){`
1	1	`    item *pt;`
3	3	`    pt = new item(someitem);`
2	2	`    assert (pt != 0);`
1	1	`    if (is_empty)`
2	2	`        {head = pt;}`
		`    else`
3	3	`         {tail->next = pt;}`
3	3	`     pt->next = NULL;`
2	2	`     tail = pt;`
		`}`
		`void list:: remove(){`
1	1	`     item *pt;`
1	1	`     item *temp;`
2	2	`     pt = head;`
1	1	`     while(pt) {`
2	2	`          temp = pt;`
3	3	`          pt   = pt->next;`
1	1	`          delete(temp);}`
3	3	`      head = tail = NULL;`
		`      cout <<"Removed all list elements"<< "\n";`
		`}`
		`boolean list::is_empty(){`
2		`      if (head == NULL)`
1		`            return(true);`
		`      else`
1		`            return(false);`
		`}`

Figure 1: Data slice profile for the member functions of the class *list*. The number of data tokens that are included in the data slice for the instance variable *head* are indicated with a number in the column with header, *head*. Similarly for the instance variable *tail*.

Data Adhesiveness is a more precise measure of the binding or relatedness among the data slices. Data Adhesiveness for a class C is defined as the ratio of the sum of the number of slices containing each glue token to the product of the number of data tokens in the class and the number of data slices. Thus,

$$DA(C) = \frac{\sum_{d \epsilon G(CSA(C))} \#\ slices\ containing\ d}{|tokens(C)| \times |CSA(C)|}$$

As an example, we apply these cohesion measures to the class *list*. The *CSA (list)* has two slices with 40 tokens and 31 glue tokens. Since there are only two data slices in the CSA(list), all glue tokens are also super-glue tokens. Hence,

$$WDC(CSA(list)) = SDC(CSA(list)) = \frac{31}{40} = 0.78$$

Data adhesiveness is given as :

$$DA(CSA(list)) = \frac{31 \times 2}{40 \times 2} = 0.78$$

A second example is given in Figure 2. In this case there are more than two slices and therefore the sets of glue and super-glue tokens are not identical. For this example, we have

$$SDC(CSA(stack)) = \frac{5}{19} = .26$$

$$WDC(CSA(stack)) = \frac{12}{19} = .63$$

$$DA(CSA(stack)) = \frac{7 \cdot 2 + 5 \cdot 3}{19 \cdot 3} = .51$$

3 Bieman and Kang's approach.

Jim Bieman and Byung-Kyoo (Benjamin) Kang treat the method and instance variable class components as the key class units that may or may not be connected [1]. A method and an instance variable are related by the way that an instance variable is used by the method. Two methods are related (connected) through instance variable(s) if both methods use the instance variable(s). Using this orientation, class cohesion can be measured by the relative connectivity (through instance variables) of the methods.

We are analyzing several alternatives for quantifying class cohesion based on:

- Directly connected methods vs. indirectly connected methods. These indirect connections can be through chains of variable uses and through messages sent between local methods.
- Read/Write dependencies (through instance variables) of methods vs. connections based only on the use (read or write) without regard to the data dependencies of instance variables.
- Local and inherited class components vs. using only locally defined components.

array	top	size	Class Stack
			`class Stack {int * array, top, size;`
			`public:`
			`Stack (int s) {`
2		2	`size = s:`
2		2	`array = new int[size];`
	2		`top = 0;}`
			`int Isempty() {`
	2		`return top==0};`
			`int Size() {`
		1	`return size};`
			`int Vtop() {`
3	3		`return array[top-1];}`
			`void Push (int item) {`
2	2	2	`if (top==size)`
			`printf("Empty stack.\n");`
			`else`
3	3	3	`array[top++]=item;}`
			`int Pop() {`
	1		`if(Isempty())`
			`printf("Full stack.\n");`
			`else`
	1		`--top;}`
			`};`

Figure 2: Data slice profile for class *stack*.

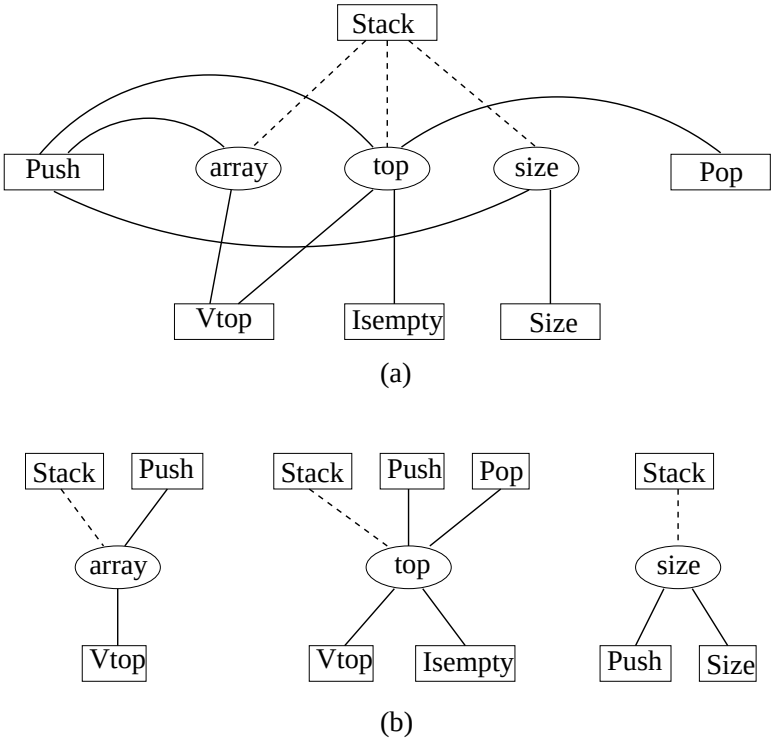

Figure 3: MIV relations for class *Stack*

We can find arguments to support any of these choices.

Individual methods in a single class can be connected via two mechanisms:

1. *MIV relations* involve communication between methods through shared instance variables. An *MIV relation* is created when two or more class methods read or write to the same class instance variable.

2. *Call relations* involve the sending of messages directly (or indirectly) from one method to another. Instance variables used by the server may also be used indirectly by the client when one method invokes another through message passing.

A call relation can be reflected by the MIV relation; two methods with a call-relation are also connected through the instance variables used by both methods. One method uses the instance variable(s) directly and the other uses the instance variable(s) indirectly through the call relation. There is no MIV relation when a server method neither writes nor reads instance variables. Thus, we do not need to include call-relations separately from MIV relations in our cohesion model.

Figure 2 shows a C++ class *Stack* and Figure 3(a) shows the MIV relations among class components of *Stack* in Figure 2. A link between a rectangle and an oval indicates that the method corresponding to the rectangle uses the instance variable corresponding to the oval. Figure 3(b) shows the connections for each instance variable. Here, the instance variable *top* is used by the methods *Stack, Push, Pop, Vtop,* and *Isempty*. All of the methods that use the variable *top* are connected through the variable *top*. These methods should be defined in one class or in classes with an inheritance relationship in order to access the instance variable.

A class constructor (e.g., method *Stack*) is an initialization function. It will generally access all instance variables in the class, and thus, share instance variables with virtually all other methods.

Constructors create connections between methods even if the methods do not have any other relationships. Thus, we remove constructor functions from our model and measures. Links between the constructor Stack and instance variables in Figure 3 are represented as dashed lines. We also do not include destructor functions in our model.

A client of a class can access only visible components of the class. Class cohesion refers to the relatedness of visible components of the class which represent its functionality. Class cohesion is the degree that those components are related. In our model of class cohesion, invisible components of a class are included only indirectly through the visible components.

Instance variables are not usually visible to the clients of a class, the state of an object is provided through class methods. In our model, cohesion of a class indicates the degree of connectivity of the visible methods in the class. In C++ programs, the visible methods are defined in "public" and "protected" sections; the methods in "private" sections are not visible. Instance variables are involved in our cohesion model through MIV relations among visible methods. Invisible methods are also involved indirectly when they are called by visible ones. Therefore class cohesion is modeled as the MIV relations among all visible methods (not including constructor or destructor functions) in the class.

3.1 Measuring Class Cohesion

The MIV relation model includes the information to define class cohesion. In our model, a method is represented as a set of instance variables directly or indirectly used by the method. A more detailed derivation of our class cohesion measures are contained in [1]. Here we summarize the derivation.

An instance variable is *directly used* by a method M if the instance variable appears as a data token in the method M. The instance variable may be defined in the same class as M or in an ancestor class of the class.

A direct/indirect call relation defines the indirect use of an instance variable. Figure 3(a) shows that methods *Size* and *Pop* are indirectly connected; *Size* is connected directly to *Push* which is in turn connected directly to *Pop*.

We define two measures of class cohesion based on the direct and indirect connections of method pairs. Let $NP(C)$ be the total number of pairs of visible methods in a class C. NP is the maximum possible number of direct or indirect connections in a class. If there are N methods in a class C, $NP(C)$ is $N*(N-1)/2$. Let $NDC(C)$ be the number of direct connections and $NIC(C)$ be the number of indirect connections in class C. The two measures are *Tight class cohesion* and *Loose class cohesion*:

1. *Tight class cohesion* (TCC) is the relative number of directly connected methods:

$$\mathsf{TCC}(C) = NDC(C)/NP(C)$$

2. *Loose class cohesion* (LCC) is the relative number of directly or indirectly connected methods:

$$\mathsf{LCC}(C) = (NDC(C) + NIC(C))/NP(C)$$

The value of LCC is always greater than or equal to the value of the corresponding TCC. For the *Stack* example of Figure 2, the class cohesion measures are:

$$\begin{aligned} \mathsf{TCC}(Stack) &= 7/10 = 0.7 \\ \mathsf{LCC}(Stack) &= 10/10 = 1 \end{aligned}$$

The TCC measure indicates that 70% of the visible methods in class *Stack* are directly related. The LCC measure shows that all visible methods of class *Stack* are related directly or indirectly.

TCC and LCC indicate the degree of connectivity between visible methods in a class. These visible methods are those defined within the class or inherited to the class. However, class cohesion measures for visible methods defined only within the class are also useful, because the measures are not affected by the cohesion of a superclass.

Local class cohesion measures are defined by using the local (non-inherited) methods in a class. The instance variables used and methods called by the visible methods for local class cohesion may include inherited variables. The local class cohesion measures for class *Stack* are equal to the class cohesion measures since class *Stack* does not use inheritance.

3.2 Relationship to Other Cohesion Measures

Bieman and Kang's approach to developing class cohesion measures is related to that used by Chidamber and Kemerer to develop the LCOM measure [5]. In [5], LCOM is defined as the number of paired methods that do not use (read or write) a common instance variable. This is most similar to a measure of class cohesion based only on direct connections. However, indirect connections are clearly an aspect of class cohesion. Consider methods A and C that are indirectly connected in this manner: method A and B reference a common variable v1, and method B and C reference a common variable v2. We cannot easily split the class containing methods A, B, and C to put method A and C into separate classes. Thus, the indirect connection does act as glue binding method A and C. Message passing can also result in a similar indirect connection. In [5], Chidamber and Kemerer focus on the similarity of methods, while we focus on the connections between the methods. Our position is that cohesion is an attribute related more to the connections that bind components than to their similarity.

Briand, Morasco, and Basili developed a measure designed to indicate cohesion in object-based systems developed using languages such as Ada [3, 4]. Their primary cohesion measure, Ratio of Cohesion Interactions (RCI), is based on the number of interactions between subroutines, variable declarations, and type declarations. Our measures are based on the connectivity between only the exported components, the methods. We do not consider references to types as connections since a method cannot affect another method through a type reference. Information must be passed between methods through a variable or a direct invocation. RCI is calculated by counting interactions rather than counting the number of variable pairs that interact. Thus, more connections are found using the RCI measure than either the LCC or TCC measures. These additional connections can represent multiple interactions between component pairs. The inclusion of interactions with type declarations seems well suited to object-based software, rather than object-oriented software where types alone are not exported. Our focus is on the connections between the key, externally visible components in object-oriented software.

4 Comparison.

Both approaches assume that it is the data connections that bind class components together. However, the two approaches use different information to compute the measurement values. Bieman and Kang look at a method as a single unit. A reference to an instance variable in only one statement in a method is as significant to the measures as if all components of the entire method affect the instance variable.

Table 1: Comparison of Class Cohesion Measures for the Stack Example

Approach	Measure	Value
Ott & Mehra	SDC	.26
	DA	.51
	WDC	.63
Bieman & Kang	TCC	.7
	LCC	1.0

In contrast, Ott and Mehra consider the degree that all method statements may affect an instance variable. For the strongest cohesion, all (or most) data tokens in all methods must affect the value of all instance variables. Thus, Ott and Mehra's class cohesion measures indicates a very "strict" view of cohesion. High cohesion classes will need to have very tightly coupled methods.

The two approaches for measuring class cohesion will clearly result in different numerical values. Bieman and Kang's cohesion measures will be higher than Ott and Mehra's measures. Table 1 compares the cohesion values for the stack example. Further analytical and empirical work is needed in order to determine which of these (or other) approaches or measures best matches our intuition about class cohesion.

5 Future Work.

We are now in the process of developing tools following both of our approaches. Our tool development makes use of the gen++/GENOA system designed by Premkumar Devanbu at AT&T Bell Labs [8]. Sufficient data is available for the analysis. We have located numerous object-oriented systems to analyze – many of these systems are in the public domain which facilitates the comparison of our work by others also working on measurements. We have already collected 20 C++ systems containing more than 53 Mbytes of C++ software.

Based on a study of the InterViews system [1], we have found that the methods in an object-oriented system exhibit high functional cohesion. The majority of the methods produce only one output. We hypothesis that one of the advantages of object-orientation is that it promotes cohesion at the method level. We also want to see if there are any empirical relationships between class cohesion and reuse and between class cohesion and faults. Initial results from the InterViews system showed that the classes that are heavily reused via inheritance exhibit lower cohesion. In general, the classes of the InterViews system demonstrated high cohesion as measured by TCC and LCC. Since these initial results about reuse are counterintuitive, we need further evidence of this relationship. Assuming we find this evidence, we need to develop an understanding of why modules with less cohesion tend to be reused more frequently.

References

[1] J.M. Bieman and B-K Kang. Cohesion and reuse in an object-oriented system. *Proc. ACM Symposium on Software Reusability (SSR'95)*, pages 259–262, April 1995.

[2] J.M. Bieman and L.M. Ott. Measuring functional cohesion. *IEEE Trans. Software Engineering*, 20(8):644–657, August 1994.

[3] L. Briand, S. Morasca, and V. Basili. Assessing software maintainability at the end of high-level design. *Proc. IEEE Conf. on Software Maintenance (CSM'93)*, September 1993.

[4] L. Briand, S. Morasca, and V. Basili. Defining and validating high-level design metrics. Technical Report CS-TR-3301, Computer Science Dept., University of Maryland, College Park, MD, 1994.

[5] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Trans. Software Engineering*, 20(6):476–493, June 1994.

[6] N. Churcher and M. Shepperd. Comments on "A metrics suite for object oriented design". *IEEE Trans. Software Engineering*, 21(3):263–265, March 1995.

[7] N. Churcher and M. Shepperd. Towards a conceptual framework for object oriented software metrics. *ACM Software Engineering Notes*, 20(2):69–76, April 1995.

[8] P. Devanbu. GENOA a customizable, language- and front-end independent code analy zer. *Proc. int. Conf. Software Engineering (ICSE)*, pages 307–317, 1992.

[9] N. Fenton. *Software Metrics - A Rigorous Approach.* Chapman and Hall, London, 1991.

[10] Bindu Mehra. Measuring data cohesion in the object-oriented paradigm. Master's thesis, Department of Computer Science, Michigan Technological University, in preparation.

[11] E. Yourdon and L. Constantine. *Structured Design.* Prentice-Hall, Englewood Cliffs, NJ, 1979.