# Cohesion and Reuse in an Object-Oriented System[*]

**James M. Bieman** and **Byung-Kyoo Kang**
Department of Computer Science
Colorado State University
Fort Collins, Colorado 80523 USA
(303) 491-7096, Fax: (303) 491-2466
bieman@cs.colostate.edu, kang@cs.colostate.edu

## Abstract

We define and apply two new measures of object-oriented class cohesion to a reasonably large C++ system. We find that most of the classes are quite cohesive, but that the classes that are reused more frequently via inheritance exhibit clearly <u>lower</u> cohesion.

## 1 Introduction

Software developers aim for systems with high cohesion and low coupling. The value of these goals has not been validated empirically [6]. Rather, they have been justified on the basis of intuition. The amount of reuse — the number of times that a component is reused — is an indicator of reusability. Of course, other factors such as the usefulness of a component are also components of reusability.

Cohesion refers to the "relatedness" of module components. 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* [7]. To apply this cohesion model to classes in object-oriented software, we need to add a new classification, *data cohesion* [5].

Bieman and Ott developed a set of functional cohesion measures based on program slices [2]. These measures apply only to individual functions; their application to entire classes is not obvious. Chidamber and Kemerer developed a Lack of Cohesion in Methods (LCOM) measure for object-oriented software [3]. LCOM is effective at identifying the most non-cohesive classes, but it is not effective at distinguishing between partially cohesive classes. LCOM indicates lack of cohesion only when, compared pairwise, fewer than half of the paired methods use the same instance variables.

Cohesion measures that are sensitive to small changes are needed in order to evaluate the relationship between cohesion and reuse. In this paper, we develop sensitive class cohesion measures and apply them to a reasonably large C++ system. We evaluate the relationship between class cohesion and private reuse in this system.

[*]Research partially supported by NASA Langley Research Center grant NAG1-1461.

Appeared in *Proc. ACM Symposium on Software Reusability (SSR'95)*, April, 1995. pp. 259-262

## 2 Class Cohesion

The components of a class are the instance variables and methods defined in the class plus those that are inherited. 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). Class cohesion is defined in terms of the relative number of connected methods in the class.

### 2.1 Relations between Class Components

Individual methods are tied together via two mechanisms. One mechanism, *MIV relations*, involves communication between methods through shared instance variables. The other mechanism, *call relations*, involves the sending of messages directly (or indirectly) from one method to another.

An *MIV relation* is created when two or more class methods read or write to the same class instance variable. We treat shared instance variables as glue that binds the class methods together.

Instance variables used by the server may also used indirectly by the client when one method invokes another through message passing. Thus, 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. Call relations can not always be determined statically, due to dynamic

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

Figure 1: An example of a class *stack*

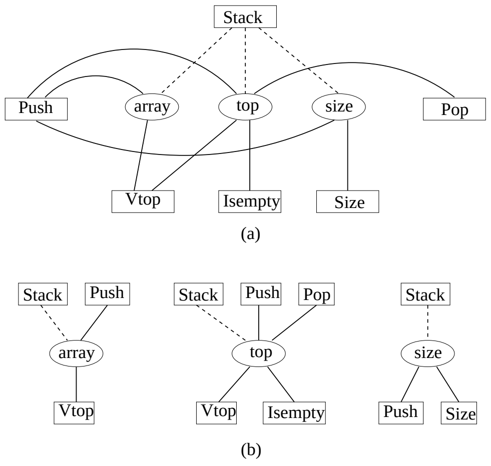

Figure 2: MIV relations for class *Stack*

binding in object-oriented software. However, we have observed very few cases where dynamic binding affects class cohesion.

Figure 1 shows a C++ class *Stack* and Figure 2(a) shows the MIV relations among class components of *Stack* in Figure 1. 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 2(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.*

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 2 are represented as dashed lines. We also do not include destructor functions in our model.

## 2.2 Visibility of Class Components

A client of a class can access only visible components of the class. Class cohesion refers 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. Therefore class cohesion is modeled as the MIV relations among all visible methods (not including constructor or destructor functions) in the class.

## 2.3 Inheritance and Cohesion

A subclass inherits methods and instance variables from its superclass(es). We have several options for evaluating cohesion of a subclass. We can (1) include all inherited components in the subclass in our evaluation, (2) include only methods and instance variables defined in the subclass, or (3) include inherited instance variables but not inherited methods. The class cohesion measures that we develop can be applied using any one of these options.

## 3 Measuring Class Cohesion

The MIV relation model includes the information to define class cohesion. A method is represented as a set of instance variables directly or indirectly used by the method. We call the representation of a method an *abstracted method*, AM.

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. $\mathsf{DU}(M)$ is a set of instance variables directly used by a method $M$.

A direct/indirect call relation defines the indirect use of an instance variable. A method $M'$ is directly called by a method $M$ if $M$ is predecessor of $M'$ in the call graph. Indirect call relations are the transitive closure of the direct call relations. Thus, a method $M'$ is indirectly called by a method $M$ if there is a path from $M$ to $M'$ in the statically determined call graph.

An instance variable is *indirectly used* by a method $M$ if (1) the instance variable is directly used by another method $M'$ which is called directly or indirectly by $M$, and (2) the instance variable directly used by $M'$ is in same object as $M$. $\mathsf{IU}(M)$ is a set of instance variables indirectly used by method $M$. An instance variable is *used* by method $M$ if the instance variable is directly or indirectly used by $M$.

A class is represented as a collection of AM's where each AM corresponds to a visible method in the class. The representation of a class is called an *abstracted class*, AC:

$$\begin{aligned} \mathsf{AM}(M) &= \mathsf{DU}(M) \cup \mathsf{IU}(M) \\ \mathsf{AC}(C) &= [\![ \mathsf{AM}(M) \mid M \in \mathsf{V}(C) ]\!] \end{aligned}$$

$\mathsf{V}(C)$ is a set of all visible methods in class $C$ and the ancestor classes of $C$. The AM's of different methods can be identical, thus there can be duplicate elements in AC. Therefore, AC is a multi-set; "$[\![$" and "$]\!]$" denote a multi-set. A *local abstracted class* (LAC) is a collection of AM's where each AM corresponds to a visible method defined only within the class:

$$\mathsf{LAC}(C) \quad = \quad [\![\mathsf{AM}(M) \mid M \in \mathsf{LV}(C)]\!]$$

$\mathsf{LV}(C)$ are the visible methods defined within class $C$. The abstracted class of the *Stack* of Figure 1 is

$$\mathsf{AC}(Stack) = \mathsf{LAC}(Stack) = [\![\{top\}, \{size\}, \{array, top\}, \{array, top, size\}, \{pop\}]\!]$$

AC(Stack) = LAC(Stack) since class *Stack* does not have a superclass.

## 3.1 Connectivity between methods

The direct connectivity between methods is determined from the class abstraction. If there exists one or more common instance variables between two method abstractions then the two corresponding methods are *directly connected.*

Two methods that are connected through other directly connected methods are *indirectly connected.* The indirect connection relation is the transitive closure of direct connection relation. Thus, a method $M_1$ is indirectly connected with a method $M_n$ if there is a sequence of methods $M_2$, $M_3$, . . ., $M_{n-1}$ such that

$$M_1 \,\delta\, M_2, \cdots, M_{n-1} \,\delta\, M_n$$

where $M_i \,\delta\, M_j$ represents a direct connection.

Figure 2(a) shows that methods *Size* and *Pop* are indirectly connected; *Size* is connected directly to *Push* which is in turn connected directly to *Pop.*

## 3.2 Definition of Measures

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 abstracted methods in $\mathsf{AC}(C)$. $NP$ 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 $\mathsf{AC}(C)$.

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

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

*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 1, 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 abstracted class (LAC) rather than the abstracted class (AC). 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.

We have built a tool that takes the class cohesion measures for C++ source programs. We specified our measures in the GENOA tool specification language [4]. We used GEN++ from AT&T Bell Laboratories, an application generator for creating code analyzers from GENOA specifications, to implement our tool.

## 4 Measuring OO reuse

We focus on private reuse—reuse within one software system [5]. We evaluate reuse from the server perspective, since this is the best orientation for evaluating reusability [1]. We are interested in two different forms of class reuse, reuse via instantiation and reuse via inheritance. A class is reused by being instantiated in other classes or by being inherited to them. *Instantiation reuse* of a class is measured as the number of classes where the class is instantiated. *Inheritance reuse* of a class is the number of classes which inherit the class, i.e., the number of descendents (both direct and indirect descendents).

## 5 Applying the Cohesion & Reuse Measures

We applied the class cohesion measures to the InterViews system, a reasonably large C++ system, developed at Stanford University. InterViews is a system for window-based applications which provides a set of classes that define the behavior of user interface objects. It consists of more than 25,000 non-commented lines of code. 14% of the classes in InterViews do not have any methods; these classes were excluded from our measurements.

We also removed all virtual methods with empty bodies. Such methods do not use instance variables or call other methods. Overloaded methods within the same class are treated as one method.

We measured instantiation reuse and inheritance reuse for all classes. We found no relationship between class cohesion and instantiation reuse in the InterViews system. However, we found significant relationships between cohesion and inheritance reuse.

The class cohesion of a subclass is affected by the class cohesion of its superclass. To remove the effects of

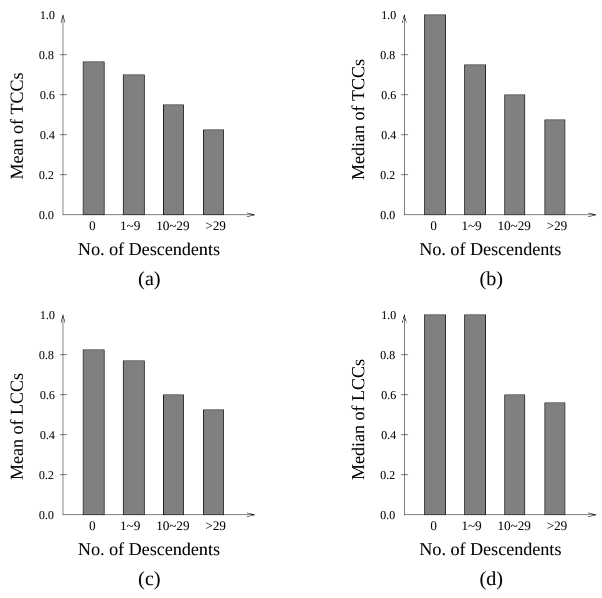

Figure 3: Number of descendents and Class Cohesion

Table 1: Depth, Descendents, & Class Cohesion

| D | No. of Descendents | N | TCC Mean | TCC Med | LCC Mean | LCC Med |
|---|---|---|---|---|---|---|
| 1 | $n = 0$ | 367 | 0.91 | 1.00 | 0.94 | 1.00 |
| 1 | $1 \leq n < 10$ | 20 | 0.69 | 0.88 | 0.75 | 1.00 |
| 1 | $10 \leq n$ | 7 | 0.58 | 0.48 | 0.61 | 0.51 |
| 2 | $n = 0$ | 190 | 0.53 | 0.61 | 0.57 | 0.71 |
| 2 | $1 \leq n < 10$ | 27 | 0.53 | 0.64 | 0.59 | 0.71 |
| 2 | $10 \leq n$ | 7 | 0.24 | 0.14 | 0.32 | 0.14 |
| 3 | $n = 0$ | 112 | 0.68 | 0.78 | 0.75 | 1.00 |
| 3 | $1 \leq n < 10$ | 28 | 0.78 | 1.00 | 0.84 | 1.00 |
| 3 | $10 \leq n$ | 6 | 0.46 | 0.52 | 0.58 | 0.70 |
| 4 | $n = 0$ | 84 | 0.81 | 1.00 | 0.90 | 1.00 |
| 4 | $1 \leq n < 10$ | 37 | 0.76 | 0.83 | 0.87 | 1.00 |
| 4 | $10 \leq n$ | 10 | 0.59 | 0.56 | 0.71 | 0.78 |
| 5 | $n = 0$ | 98 | 0.72 | 0.86 | 0.79 | 1.00 |
| 5 | $1 \leq n < 10$ | 31 | 0.70 | 0.69 | 0.79 | 1.00 |
| 5 | $10 \leq n$ | 6 | 0.67 | 0.85 | 0.67 | 0.85 |
| 6 | $n = 0$ | 85 | 0.77 | 0.87 | 0.81 | 1.00 |
| 6 | $1 \leq n < 10$ | 17 | 0.75 | 0.74 | 0.79 | 0.85 |
| 6 | $10 \leq n$ | 5 | 0.50 | 0.52 | 0.52 | 0.60 |
| 7 | $n = 0$ | 35 | 0.84 | 1.00 | 0.87 | 1.00 |
| 7 | $1 \leq n < 10$ | 18 | 0.67 | 0.71 | 0.75 | 0.82 |
| 7 | $10 \leq n$ | 2 | 0.08 | 0.17 | 0.08 | 0.17 |
| 8 | $n = 0$ | 51 | 1.00 | 1.00 | 0.84 | 1.00 |
| 8 | $1 \leq n < 10$ | 11 | 0.72 | 0.75 | 0.81 | 0.82 |
| 9 | $n = 0$ | 14 | 0.86 | 1.00 | 0.93 | 1.00 |

*D* is the depth in the inheritance hierarchy.

the superclass, we use only local cohesion—we do not include inherited methods in our measurement.

Figure 3 shows the relationship between the number of descendents and local class cohesion. Average values of tight class cohesion and loose class cohesion are provided for four different categories based on the number of descendents. Figure 3 clearly shows that the classes that are reused more frequently exhibit lower cohesion. Table 1 shows that this relationship holds generally for all levels of depth in the inheritance hierarchy.

We applied a T-test and the Wilcoxon rank-sum test to evaluate the significance of our results. A T-test can be used for data with a normal distribution and an interval scale, and the Wilcoxon rank-sum test can be used if there is a question concerning distributions or if the data is ordinal. Both tests shows that the relationship we see in Figure 3 and Table 1 is significant (to the .05 level) and not due to chance.

## 6 Discussion of Results

If a class is designed in ad hoc manner and unrelated components are included in the class, the class represents more than one concept and does not model an entity. A class designed so that it is a model of more than one entity will have more than one group of connections in the class. The cohesion value of such a class is likely to be less than 0.5. For example, if five of the six methods in a class are connected and the remaining method has no connections, the $TCC$ and $LCC$ of the class are both 0.67. If three of the six methods in a class are connected, and the other three are also connected with no connection between those two groups, both the $TCC$ and $LCC$ of the class are 0.40. Therefore, the class cohesion measures can be used to locate the classes that may have been designed inappropriately.

Most of the classes in InterViews are quite cohesive. The mean $TCC$ is 0.75 and median is 1.0; the $LCC$ mean is 0.8 and median is 1.0. The classes have an average of about six local visible methods. If the $LCC$ of a class is 0.8 and the class has six local visible methods, then 80% of the pairs of methods in the class are connected, i.e., among 15 pairs of methods 12 pairs are connected. Thus, we know that most of InterViews classes are quite cohesive and were designed carefully.

Our results show that the classes that are heavily reused via inheritance exhibit lower cohesion. We expected to find that the most reused classes would be the most cohesive ones. Studies of additional software systems are needed to confirm these results.

## References

[1] J. Bieman. Deriving measures of software reuse in object-oriented systems. *Proc. BCS-FACS Workshop Formal Aspects of Measurement*, pp. 79-82. Springer 1992.

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

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

[4] P. Devanbu. GENOA a customizable, language- and front-end independent code analyzer. *Proc. ICSE-14*, pp. 307–317, 1992.

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

[6] N. Fenton, S.L. Pfleeger, and R. Glass. Science and substance: a challenge to software engineers. *IEEE Software*, 11(4):86–95, July 1994.

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