A Practical Look at the Lack of Cohesion in Methods Metric

Letha Etzkorn, Carl Davis, and Wei Li

The University of Alabama in Huntsville

letzkorn@cs.uah.edu

Software metrics for the procedural software development paradigm have been extensively studied. Metrics such as McCabe's cyclomatic complexity metric[1] and Halstead's Software Science metrics[2] are well known and frequently used to measure software complexity in the procedural paradigm. More recently, software metrics that are tailored to the measurement of design complexity in the object-oriented paradigm have been developed. Chidamber and Kemerer proposed a draft suite of software metrics for object-oriented software in 1991[3] that included six object-oriented design metrics based on measurement theory. This suite of metrics included depth of the inheritance tree (DIT), number of children (NOC), coupling between objects (CBO), response for a class (RFC), weighted methods per class (WMC), and lack of cohesion of methods (LCOM).

The LCOM metric has been subject to multiple interpretations which can greatly influence the LCOM value derived for a particular class. This article compares and analyzes the definition and implementation variations of the LCOM metric, and provides an assessment of this metric.

DEFINITION OF THE LCOM METRIC

The multiple definitions of the LCOM metric currently in use include: 1) the original definition of the LCOM metric by Chidamber and Kemerer[3] 2) the definition of the LCOM metric provided by Li and Henry[4,5], 3) the redefinition of the Li and Henry version of the LCOM metric by Hitz and Montazeri[6], and 4) the redefinition of their original LCOM metric by Chidamber and Kemerer.[7,8]

Chidamber and Kemerer original OOPSLA definition

Chidamber and Kemerer originally defined the Lack of Cohesion in Methods metric (LCOM) as follows[3]: Consider a Class C1 with methods M1, M2,..., Mn. Let { I_i }= set of instance variables used by method M_i.

There are n such sets {I1}, ... {In}. LCOM = the number of disjoint sets formed by the intersection of the n sets.

The viewpoints listed for this metric are:

1) Cohesiveness of methods within a class is desirable, since it promotes encapsulation of objects.

2) Lack of cohesion implies classes should probably be split into two or more sub-classes.

3) Any measure of disparateness of methods helps identify flaws in the design of classes.

4) Low cohesion increases complexity, thereby increasing the likelihood of errors during the development process.

Li and Henry definition

The definition of disjointness of sets given in the Chidamber and Kemerer OOPSLA '91 paper was somewhat ambiguous, and was further defined by Li and Henry:[4]

LCOM = number of disjoint sets of local methods; no two sets intersect; any two methods in the same set share at least one local instance variable; ranging from 0 to N; where N is a positive integer. Li et al. demonstrated a calculation of their definition of the LCOM metric using the example shown again here in Figure 1[5]. This example is used here to compare different implementations of the LCOM metric (see Figure 3).

Chidamber and Kemerer redefinition

Chidamber and Kemerer, in their 1994 paper,[6,7,8] redefined the LCOM metric. Their new definition of the LCOM metric is as follows:

Consider a Class C_1 with n methods M_1, M_2, ..., M_n. Let $\{ I_j \}$ = set of instance variables used by method M_i.

There are n such sets $\{ I_1 \}$ ... $\{ I_n \}$. Let P = $\{ (I_i, I_j) \mid I_i$ intersecting I_j is equal to the null set $\}$ and Q = $\{ (I_i, I_j) \mid I_i$ intersecting I_j is not equal the null set$\}$. If all n sets $\{ I_1 \}$...$\{ I_n \}$ are the null set then let P = the null set.

LCOM = $|P| - |Q|$, if $|P| > |Q|$

= 0 otherwise

In this definition, Q = the number of pairs of methods which both access the same (one or more) instance variables. P = the number of pairs of methods which do not have any instance variable accesses in common. As described by Basili et al.[6], this definition of

```
class Location {protected:
    int X,Y;
public:
   Location (int InitX, int InitY) { X=InitX; Y=InitY; }
   int GetX() { return X;}
   int GetY() { return Y;}
};

class Point: public Location {
protected:
   Boolean Visible;
public:
   Point(int InitX, int InitY);
   virtual void Show();
   virtual void Hide();
   virtual void Drag(int DragBy);
   Boolean IsVisible() {return Visible;}
   void MoveTo( int NewX, int NewY);
};

class Circle: public Point {protected:
   int Radius;
public:
   Circle(int InitX,int InitY,int InitRadius);
   void Show();
   void Hide();
   void Expand(int ExpandBy);
   void Contract(int ContractBy);
};

// The following definition may be stored in another file

Point::Point(int InitX, int InitY): Location (InitX, InitY){
  Visible = false;
}

void Point::Show() {
  Visible = true;
  putpixel(X,Y, getcolor());
}

void Point::Hide() {
  Visible = false;
  putpixel(X,Y,getbkcolor());
}

void Point::MoveTo(int NewX, int NewY)  {
   Hide();
   X=NewX;
   Y=NewY;
   Show();
}

void Point::Drag(int DragBy) {
  int DeltaX, DeltaY;
  int FigureX, FigureY;

  Show();
  FigureX = GetX();
  FigureY = GetY();

  while (getdelta(DeltaX, DeltaY)) {
     FigureX = FigureX + (DeltaX * DragBy);
     FigureY = FigureY + (DeltaY * DragBy);
     MoveTo( FigureX, FigureY );  }
}

Circle::Circle(int InitX, int InitY, int InitRadius) : Point(
InitX, InitY) {
  Radius = InitRadius; }

void Circle::Show() {
  Visible = true;
  circle(X,Y,Radius); // this is an external function call,
not a constructor call
}

void Circle::Hide() {
  unsigned int TempColor;

  TempColor = getcolor();
  setcolor(getbkcolor());
  Visible=false;
  circle(X,Y,Radius);
  setcolor(TempColor);
}

void Circle::Expand(int ExpandBy) {

  Hide();
  Radius = Radius + ExpandBy;
   if (Radius < 0)
      Radius = 0;

  Show();
}
void Circle::Contract(int ContractBy) {
  Expand(-ContractBy);
}
```

Figure 1. A C++ Example Used to Illustrate Different LCOM Interpretations

LCOM is the number of pairs of member functions without shared instance variables, minus the number of pairs of member functions with shared instance variables. However, the metric is set to zero whenever the above subtraction is negative.

This definition of LCOM has some drawbacks in that classes with widely different cohesions will result in the same LCOM value. Chidamber and Kemerer themselves note that "the LCOM metric for a class where |P| = |Q| will be zero. This does not imply maximal cohesiveness, since within the set of classes with LCOM = 0, some may be more cohesive than others". Basili et al.[6] performed a statistical study where LCOM, calculated in this manner, was insignificant in their analysis of fault detection. The definition of LCOM was not appropriate in their case since it set cohesion to zero for classes with very different cohesions and kept them from analyzing the actual impact of cohesion on their data sample. Hitz and Montazeri[7] discussed several cases where LCOM, calculated in this manner, would result in various anomalies when attempting to predict cohesion.

Hitz and Montazeri definition

Hitz and Montazeri proposed a different, graph-theoretic formulation of Li and Henry's version of the LCOM metric:

Let X denote a class, I_X the set of instance variables of X, and M_X the set of its methods. Consider a simple, unidirected graph $G_X(V,E)$ with $V = M_X$, and E = {<m,n> is an element of V X V| for every i that is an element of I_X : (m accesses i) intersects (n accesses i) }, i.e., exactly those vertices are connected which represent methods with at least one common instance variable. LCOM(X) is then defined as the number of connected components of G, that is, the number of method "clusters" operating on disjoint sets of instance variables. They claim that this new formulation is equivalent to Li and Henry's definition of LCOM.

LCOM calculation examples/comparisons

During the rest of this paper, the notations defined in the following table will be used for the various LCOM definitions:

LCOM1	Chidamber and Kemerer revised definition
LCOM2	Li and Henry definition

Table 1.

To clarify the difference between some of the various definitions of LCOM, the following examples are provided:

<u>Example 1:</u>

Given:

member variables: I,J,K,L

member functions: A,B,C,D

Member function A accesses variables {I,L}.

Member function B accesses no variables.

Member function C accesses variables {J,L}.

Member function D accesses variable {K}.

Using LCOM2, the disjoint sets of methods, where any two methods in the same set share at least one local instance variable, would be:

{**A**,**C**}, {**B**}, {**D**}

The LCOM2 value would then be 3, which would be derived from counting the number of sets.

Using LCOM1:

A intersecting **B** = null set

A intersecting **C** = { **L** }

A intersecting **D** = null set

B intersecting **C** = null set

B intersecting **D** = null set

C intersecting **D** = null set

P = count of the intersections whose result is the null set = 5.

Q = count of the intersections whose result is not null = 1

LCOM1 = |P| - |Q| = 5 - 1 = 4

(since |P| > |Q|)

Another difference between the two metrics is shown in the following example:

<u>Example 2:</u>

Given:

Member variables: I,J,K,L.

Member functions: A,B,C,D.

Member function A accesses variable {I}.

Member function B accesses variable {I}.

Member function C accesses variable {I}.

Member function D accesses variables

{I,J,K,L}.

Using LCOM2:

The set for LCOM2 is:

{A, B, C, D}

and therefore LCOM2 = 1.

Using LCOM1:

A intersecting B = {I}

A intersecting C = {I}

A intersecting D = {I}

B intersecting C = {I}

B intersecting D = {I}

C intersecting D = {I}

Thus P = count of the intersections whose result is the null set = 0.

Q = count of the intersections whose result is not null = 6.

Therefore LCOM1 = 0 (since $|P| < |Q|$).

Thus a class that is perfectly cohesive, measured by LCOM2, would have a value of 1, whereas the same class would have a value of 0 when measured by LCOM1.

Consider the case of a perfectly uncohesive class. The value of LCOM2 would equal the number of member functions in the class. Thus completely uncohesive classes consisting of larger number of member functions would have a larger LCOM2 value than completely uncohesive classes consisting of a smaller number of member functions. This is reasonable, since the cohesion is indeed worse -- a class providing more unrelated functionalities can be considered less cohesive than a class providing fewer unrelated functionalities.

In the case of a completely uncohesive class, LCOM1 would have a value equal to n taken two at a time, where n = the number of member functions in the class. Thus this metric also results in larger values for LCOM1 for uncohesive classes with larger number of member functions. One special case is the treatment of a class whose member functions do not access any of the class' member variables. The LCOM1 definition (revised Chidamber and Kemerer definition of LCOM) says that if all n sets $\{ I_1 \}...\{ I_n\}$ are the null set then let P = the null set. Thus $|P| = 0$, and thus LCOM1 = 0. This could tend to cause confusing results. Consider, for example, a class that contains one member variable and 5 member functions that do not access that variable (the variable is unused).

Thus, the LCOM1 for that class = 0. If one of the functions accesses that variable, the LCOM1 for the class becomes 5 taken two at a time (5 choose 2), which is 10. Yet both classes are still completely uncohesive in that they have no member functions that share any member variables. With the LCOM2 definition (Li and Henry definition), both classes would have the same LCOM value.

LCOM IMPLEMENTATION

There are some variations on the LCOM metric that are independent of which definition is used. Some of these variations include the determination of exactly which member variables and member functions take part in the calculations.

Inclusion of inherited variables

One question of LCOM implementation is whether or not inherited member variables should be used as part of the cohesiveness determination. Neither of the Chidamber and Kemerer definitions specifies whether or not inherited variables should be used. The Li and Henry definition specifies local variables only. Consider the case of the C++ example given in Figure 1. The Point::Hide( ) member function and the Point::Show( ) member function would be considered as having no member variables in common by the definition of the Li and Henry LCOM metric (LCOM2). However, both of these functions use the X, Y coordinates inherited from the class Location. The Hide member function hides a pixel (rewrites the pixel in background color) whose location is given by X,Y coordinates. The Show member function shows a pixel (rewrites the pixel in the current color) whose location is given by X,Y coordinates. Obviously these are related graphics routines, and should form part of the same class. Thus there is an argument in favor of the use of inherited variables in the LCOM2 metrics calculation.

Inclusion of constructor or destructor functions

Another possible problem with the current implementations of the LCOM metric is the inclusion of the constructor member function and/or the destructor member function in the LCOM metrics calculation. Constructor member functions, for example, tend to

```
Class example  {
        int Visible;
        int X,Y;
        float salary;
        int val_from_port;

        example( );  // Constructor
        void Hide( );
        void calculate_salary(int num_months, float
                                   amt_per_month);
        void read_val_from_input_port( );
}

void example::example( ) {
          Visible = false;
          salary = 0.0;
          val_from_port = 0;}
}
```

```
void example::Hide( ) {
           Visible = false;
            putpixel(X,Y, getbkcolor() );
}

void calculate_salary(int num_months, float
                                   amt_per_month) {
           salary = num_months * amt_per_month;
}

void read_val_from_input_port( ) {
           val_from_port = inportb(PORTA);
}
```

Using LCOM1,
P = 3, Q = 3, so LCOM1 = |P| - |Q| = 0, which supposedly means the class is completely cohesive

Using LCOM1, but leaving out the constructor,
P = 3, Q = 0, so LCOM1 = |P| - |Q| = 3, which is a more reasonable cohesiveness value, since the class is actually not cohesive

Using LCOM2,
the LCOM2 sets are { example( ) (constructor), Hide( ), calculate_salary( ), read_val_from_input_port( ) }
so LCOM2 = number of disjoint sets = 1, which supposedly means the class is completely cohesive

Using LCOM2, but leaving out the constructor,
the LCOM2 sets are: {Hide( ) }
{calculate_salary( ) }
{read_val_from_input_port ( ) }
so LCOM2 = number of disjoint sets = 3, which is a more reasonable cohesiveness value, since the class is actually not cohesive

Figure 2. A C++ Constructor Example

include all or most of the member variables. This is reasonable, since a primary purpose of a constructor function is to initialize the member variables of a class. Consider the class Location, shown in Figure 1. The Location class initializes both the X and Y member variables, which is reasonable. However, this results in LCOM2 = 1 for the

LCOM2 (Li and Henry Definition of LCOM)

	LCOM2		LCOM2	
	considering inherited variables		not considering inherited variables	
Class	with constructor	without constructor	with constructor	without constructor
Location	1	2	1	2
Point	2	2	3	3
Circle	2	2	2	2

LCOM1 (Revised Chidamber and Kemerer Definition of LCOM)

	LCOM1		LCOM1	
	considering inherited variables		not considering inherited variables	
Class	with constructor	without constructor	with constructor	without constructor
Location	0	1	0	1
Point	0	0	3	4
Circle	0	0	0	0

Figure 3. LCOM Metrics for code in Figure 1

class (see Figure 3). The LCOM2 set of functions that access the same set of member variables would be:

{Location, GetX, GetY}

If the constructor function were not included in the metrics calculation, then LCOM2 would = 2, and the LCOM2 sets of functions that access the same set of member variables would be:

{GetX }, {GetY}

Consider the same example using LCOM1. In the first case, where the constructor function is included, we have:

Location( ) intersects GetX( )

Location( ) intersects GetY( )

GetX( ) intersects GetY( )

Thus P = 0, Q = 3, so LCOM1 = 0 since |P| < |Q|.

In the second case, where the constructor function is not included, we have:

GetX( ) intersects GetY( )

thus P = 0, Q = 1, so LCOM1 = 0 since |P| < |Q|.

In this particular case, LCOM1 did not vary based on whether the constructor or destructor was included or not. However, if this example is extended, then any class which possesses a constructor that initializes all variables will be considered as perfectly cohesive, even if the other methods have no variables at all in common. See Figure 2 for an example of such a class.

Often the destructor function operates in a similar manner to the constructor function in that it accesses most or all of the variables of a class. This occurs less often, however, since a destructor function is more commonly used to return pointer values to the heap. Integer variables are seldom accessed using a destructor function. Also, it is very common for a class to have a constructor function, but not to have an explicit destructor function. In a package where constructor functions are common, but destructor functions are rare, the destructor function may tend to acquire a greater importance to the code. Thus there is a good argument for not including the constructor function in the LCOM calculation, but still continuing to include the destructor function in the calculation.

Comparison of LCOM implementations for Li example in Figure 1

Since the example in Figure 1 was used by Li et al. to demonstrate the calculation of the LCOM2 metric, a comparison of the different possible implementations of the LCOM metric for the code shown in Figure 1 is instructive. Figure 3 contains the values calculated for the different implementations of the LCOM metric for this example.

The differences between the various LCOM metrics for the class Location is obviously not due to the use of inherited variables in the metrics calculation, since Location is a base class. However, LCOM does vary depending on whether or not the constructor function is used in the metrics calculation. The constructor function here performs a function similar to that shown in Figure 2; that is, it initializes all variables in the class.

The class Point shows no difference between calculations of the LCOM2 metric with or without the constructor function. However, both the Point and Circle classes represent simple cases where the constructor initialization problem would probably not be obvious -- in both classes there is only a single member variable. The Circle class shows no difference between the LCOM metrics calculated considering inherited variables, and the LCOM metrics calculated not considering inherited variables. As it happens, the only inherited variables accessed by Circle member functions occur in member functions that access the same local variable (Show and Hide). The Point class shows a definite difference between the LCOM metrics calculated with and without considering inherited variables. The difference is due to the MoveTo member function. MoveTo does not access the local member variable "Visible". However, it does access the inherited variables X and Y. Notice that in this case, the differently calculated cohesion values should probably be fairly similar, since the difference in the calculations is based on a single member function out of six member functions total (5 member functions total if not including the constructor function). This leads to a pair of interesting observations, discussed below:

The LCOM2 implementations result in the following disjoint sets:

LCOM2 with inheritance, with constructor:

{constructor Point( ), Show( ), Hide( ), IsVisible( ), MoveTo( ) }

{Drag}

LCOM2 with inheritance, no constructor:

{Show( ), Hide( ), IsVisible( ), MoveTo( )}

{Drag}

LCOM2, no inheritance, with constructor:

{constructor Point( ), Show( ), Hide( ), IsVisible( ) }

{Drag}

{MoveTo}

LCOM2, no inheritance, no constructor:

{Show( ), Hide( ), IsVisible( ) }

{Drag( )}

{MoveTo( )}

Thus the MoveTo member function results in the addition of only one extra disjoint set in any case, and the LCOM2 metric is only one value greater.

LCOM1 varies more between the two implementations (with or without inherited variables) than does LCOM2.

LCOM1 with inheritance, with constructor:

P = 7, Q = 8

LCOM1 with inheritance, no constructor:

P = 5, Q = 5

LCOM1, no inheritance, with constructor:

P = 9, Q = 6

LCOM1, no inheritance, no constructor:

P = 7, Q = 3

The P and Q values have a wide variation, as does the metric itself. Thus a change of only a single member function can result in greatly different cohesiveness values of LCOM1.

Which LCOM Metric Best Measures Cohesion?

Since the different definitions and implementations of the LCOM metric can result in different values for LCOM, the question is which definition and which implementation of LCOM best measures cohesion?

Various C++ classes and hierarchies of classes were chosen from three independent C++ GUI packages. Alternative versions of the LCOM metric for these classes were calculated using the **PATR**icia system[9,10,11] (**P**rogram **A**nalysis **T**ool for **R**euse). Seven highly experienced domain experts subjectively rated each class for cohesiveness by categorizing it as acceptably cohesive, or not cohesive. An acceptably cohesive class was classed as 100%, a non-cohesive class as 0%. An attempt had been made earlier to rate classes on a tighter scale; however, it was found that classes that the experts agreed had the same cohesion (should be broken into an agreed upon number of sub-classes) was rated as "fair" by some experts, or "poor" by others -- that there was no true agreement on scale. Thus it was found that the rougher measure was more appropriate in this case.

The various LCOM values produced by the **PATR**icia system for each class were compared to the averaged cohesiveness ratings of the experts. LCOM was measured in 8 different ways: 1) revised Chidamber and Kemerer definition (LCOM1), including inherited variables, including constructor function, 2) revised Chidamber and Kemerer definition (LCOM1), including inherited variables, not including constructor function 3) revised Chidamber and Kemerer definition (LCOM1), not including inherited variables, including constructor function 4) revised

	Chidamber and Kemerer revised definition LCOM1				**Li and Henry definition LCOM2**			
Class	with Inheritance with Const.	with Inheritance without Const.	without Inheritance with Const.	without Inheritance without Const.	with Inheritance with Const.	with Inheritance without Const.	without Inheritance with Const.	without Inheritance without Const.
1	34	34	34	34	8	8	8	8
2	24	16	24	16	6	5	6	5
3	343	321	343	321	24	24	24	24
4	287	276	287	276	18	18	18	18
5	247	247	247	247	21	21	21	21
6	1	0	1	0	2	1	2	1
7	1142	1118	1142	1118	37	44	37	44
8	195	166	435	406	15	14	30	29
9	321	298	323	298	24	24	25	24
10	64	48	136	120	9	8	17	16
11	22	15	28	15	6	6	8	6
12	48	31	66	49	7	6	10	9
13	76	66	76	66	10	10	10	10
14	25	32	91	78	4	5	14	13
15	28	21	28	21	8	7	8	7
16	378	351	378	351	27	26	27	26
17	13	10	13	10	5	5	5	5
18	2	1	4	3	2	2	3	3

Figure 4. Difference in Values of Different LCOM Implementations

Chidamber and Kemerer definition (LCOM1), not including inherited variables, not including constructor function 5) Li and Henry definition (LCOM2), including inherited variables, including constructor function 6) Li and Henry definition (LCOM2), including inherited variables, not including constructor function 7) Li and Henry definition (LCOM2), not including inherited variables, including constructor function 8) Li and Henry definition (LCOM2), not including inherited variables, not including constructor function.

Analysis of LCOM Numeric Values

Each version of LCOM, collected for each class, is shown in Figure 4. There are certain interesting aspects to these results. First, consider class number 7. This class results in large values for both LCOM1 and LCOM2, so presumably the class possesses a large number of non-cohesive member functions. However, consider the values for LCOM1. The largest number occurs in the implementation of LCOM1 that does not use inheritance, and that does include the constructor function. In this case the value is 1142. This value is well over twice as large as the next largest value for that implementation, which is 435. This occurs because LCOM1 is bounded by the number of combinations of two functions possible in the number of member functions, which is:

$$n! / [(2!) * (n-2)!]$$

This can result in large values of LCOM1 for very non-cohesive functions with large numbers of member variables, whereas most other classes will have very much smaller values of LCOM1.

Now consider classes 8 and 9. When inheritance is included in the calculation, class 8 is considered to be more cohesive than class 9 by both LCOM1 and LCOM2, irregardless of whether or not the constructor function is included in the calculation. However, when inheritance is not considered in the calculation, class 8 shows as less cohesive than class 9. Similarly for classes 13 and 14.

For LCOM2, no changes in rank of cohesiveness of classes were found when comparing implementations that did not consider the constructor function versus those

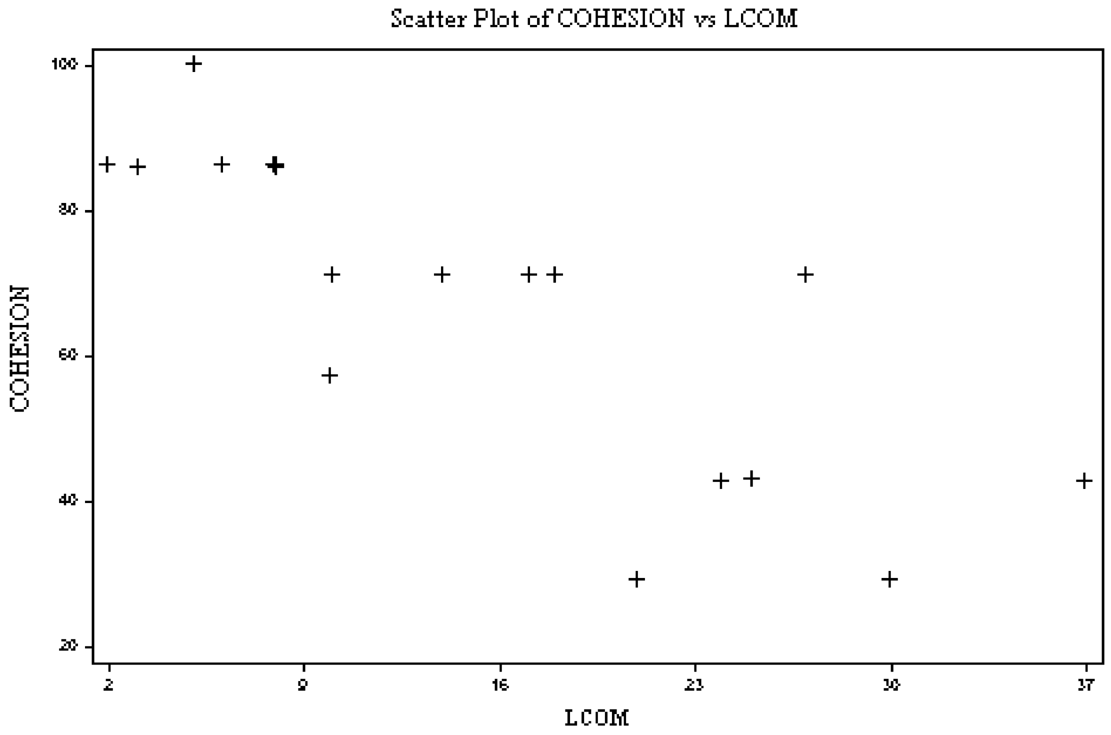

Figure 5. LCOM2 (Li and Henry LCOM), without inheritance, with

that did consider the constructor function, although in several cases the relative distance between values in a comparison of one class versus another did vary. One such variation can be found when comparing class 1 to class 2. When considering the implementations that included the constructor function versus those that did not include the constructor function, the relative values of LCOM2 of the two classes varied.

For LCOM1, a change in rank of cohesiveness was found when comparing implementations that did not consider the constructor function versus those that did consider the constructor function. This change in rank is in classes 14 and 15. With the constructor function, class 14 shows as more cohesive than class 15. Without the constructor function, class 14 shows as less cohesive than class 15.

Linear Regression Analysis

A linear regression study was performed

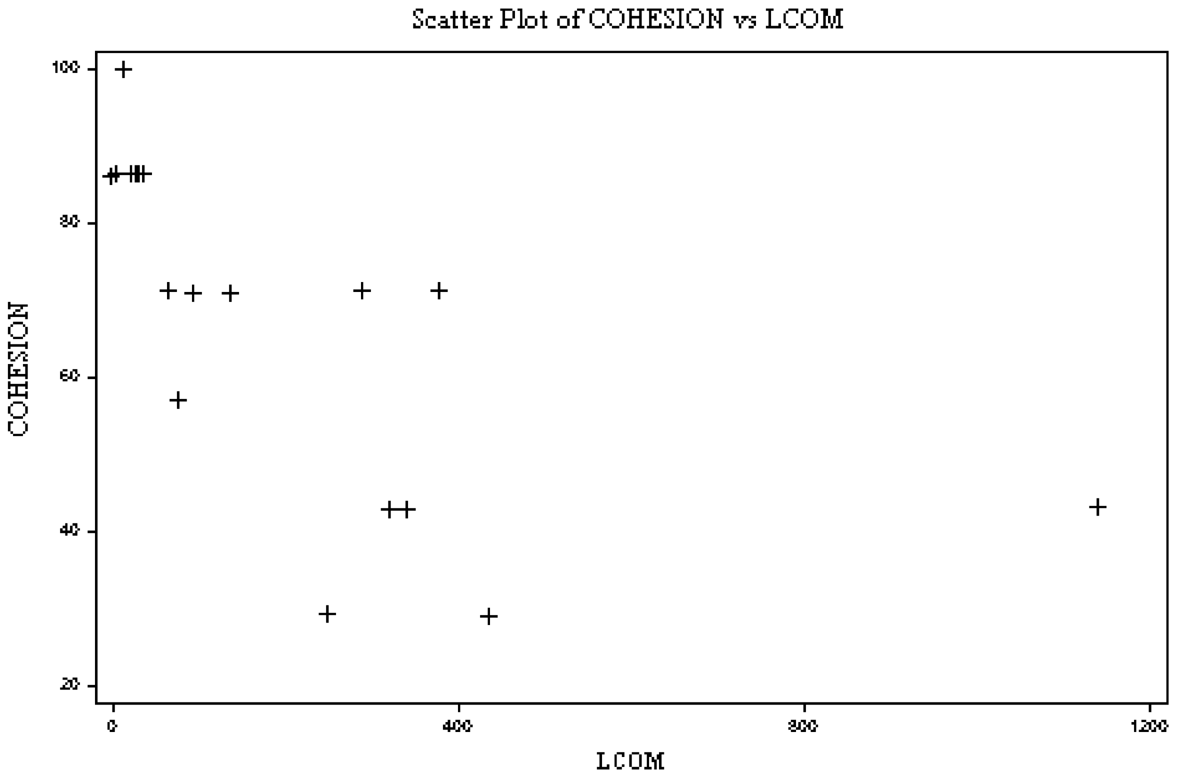

Figure 6. LCOM1 (Chidamber and Kemerer revised LCOM), without inheritance, with constructor

comparing the experts' ratings of cohesiveness to the various implementations of the LCOM metric. This was a simple study with LCOM as the independent variable, and cohesiveness (as measured by the experts) as the dependent variable. For LCOM2 the best results were obtained using the LCOM2 implementation that did not include inheritance, and that did include the constructor function. In this case the regression was highly significant ($p < 0.0001$), and the R^2 value was 66%. R^2 for LCOM2, without inheritance and without the constructor was 62%. R^2 for LCOM2, with inheritance, was 49% with the constructor, and 46% without the constructor. A scatter plot of LCOM2 without inheritance, and with constructor, is shown in Figure 5.

Considering only LCOM2, these results were somewhat surprising in view of the anomalies discussed earlier relating changes in rank between classes measured with inheritance considered, and without inheritance considered. A possible reason for this might be that inherited variables are used as general purpose variables such as global display flags. The use of such variables would not be considered by the experts as showing a relationship between member functions. The difference between the LCOM2 implementations with the constructor, and without the constructor was very small.

For LCOM1, the best results were obtained using the LCOM1 implementation that did not include inheritance, and that did include the constructor, although the difference from LCOM1 measured without inheritance but also without the constructor was minimal. However, LCOM1 had an R^2 only of 41% in the best case ($p < 0.0043$). R^2 was 30% in the worst case (with inheritance, without constructor). A scatter plot of the best case (without inheritance, with constructor) is shown in Figure 6.

SUMMARY

Several different definitions of LCOM exist. Different implementations of each of these definitions, either employing inheritance or not employing inheritance, employing the constructor function, or not employing the constructor function, are possible.

The Li and Henry definition of LCOM (LCOM2), which did not include inherited variables, and that did include the constructor function in the calculations correlated well with the expert's determination of cohesiveness.

The revised Chidamber and Kemerer metric (LCOM1) has several problems. First, classes of widely different cohesions are counted as having LCOM1 = 0. This corresponds to findings by Basili et al.[6], where this definition of LCOM set cohesion to zero for classes with very different cohesions. Second, some classes with the same cohesion (totally uncohesive) can receive different LCOM1 values. This is similar to a finding by Hitz and Montazeri[7] that in some cases classes with the same cohesion can have different LCOM1 values. Hitz and Montazeri found several cases in which classes with a cohesion of two (classes that should be subdivided into two other classes) had different LCOM1 values. Third, the range of the LCOM1 metric is limited by (n choose 2), where n is the number of member functions in the class. This can result in certain uncohesive classes having extremely large values for LCOM1, that are very much larger than the usual range for LCOM1. This study was not able to show a large amount of correlation between this LCOM metric (LCOM1) and cohesiveness.

The Li and Henry LCOM definition (LCOM2) gives classes with different cohesions a different LCOM2 value. The range of the LCOM2 metric is limited by the number of member functions in the class, which is usually quite a small number. A good correlation between the Li and Henry LCOM metric (LCOM2) and cohesion has been demonstrated.

References

1. McCabe, T.J. A complexity measure, IEEE TRANSACTIONS ON SOFTWARE ENGINEERING 2(4): 308-320, 1976.

2. Halstead, M.H. ELEMENTS OF SOFTWARE SCIENCE, Elsevier North-Holland, New York, 1977.

3. Chidamber, S.R., and C.F. Kemerer. Towards a metrics suite for object-oriented design, PROCEEDINGS: OOPSLA '91, July 1991, pp. 197-211.

4. Li, W. and S. Henry. Maintenance metrics for the object-oriented paradigm, PROCEEDINGS OF THE FIRST INTERNATIONAL SOFTWARE METRICS SYMPOSIUM, May 21-22, 1993, pp. 52-60.

5. Li, W., S. Henry, D. Kafury, and R. Schulman. Measuring object-oriented design, JOURNAL OF OBJECT-ORIENTED PROGRAMMING, JULY/AUGUST 1995, pp. 48-55.

6. Hitz, M. and B. Montazeri. Chidamber and Kemerer's Metrics Suite: a measurement theory perspective, IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, 22(4), April 1996, pp. 267-271.

7. Basili, V., L. Briand, and W.L. Melo, A validation of object-oriented metrics as quality indicators, IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, 22(10), October 1996, pp.751-761.

8. Chidamber, S.R., and C.F. Kemerer. A metrics suite for object-oriented design, IEEE TRANSACTIONS ON SOFTWARE

ENGINEERING, 20 (6) June 1994, pp. 476-493.

9. Etzkorn, L.H., and C.G. Davis. Automated object-oriented reusable component identification, KNOWLEDGE-BASED SYSTEMS, Elsevier Science, Ltd., Oxford, England (accepted).

10. Etzkorn, L.H., C.G. Davis, L.L. Bowen, D.B. Etzkorn, L.W. Lewis, B.L. Vinz, and J.C. Wolf, A Knowledge-based approach to object-oriented legacy code reuse, PROCEEDINGS OF THE SECOND IEEE INTERNATIONAL CONFERENCE ON ENGINEERING OF COMPLEX COMPUTER SYSTEMS, Montreal, Canada, Oct. 21-25, 1996, pp. 493-496.

11. Etzkorn, L.H., A metrics-based approach to the automated identification of object-oriented reusable components: a short overview. OOPSLA '95 Doctoral Symposium, Austin, TX, October 16-19, 1995.