Observations on Lack of Cohesion Metrics

Sami Mäkelä and Ville Leppänen

Abstract: *Lack of Cohesion Metric (LCOM) is perhaps the most used metric when trying to measure the goodness of a class written in some object-oriented language. We apply the basic LCOM metric to the Java SDK 5.0 and Eclipse libraries. LCOM gives a bad value for most of the classes. We study the reasons and characterize sets of classes for which one should not apply LCOM to determine the goodness of an implementation. Yet, one of the major reasons for bad LCOM value is observed to be its dependence on the measured class size. Normalization in this respect is considered.*

Key words: *Cohesion, LCOM, metrics, object-oriented programming, JSDK, Eclipse.*

1 INTRODUCTION

One important feature for object-oriented programs is *cohesion*. A class should represent a single logical concept, and not be a collection of miscellaneous features. The purpose of a cohesion value is to tell how well a class fulfils this design principle.

We study the cohesion metric LCOM (by Chidamber and Kemerer) [4,5,6] of object-oriented programs. We apply metrics for the classes of Java SDK 5.0 and Eclipse library. The basic cohesion metric is shown to report rather bad results about the Java library. In our opinion, the results do not reflect the goodness of the library -- the problem rather is that the basic cohesion metric only suits to certain kind of classes.

The LCOM metric is based on the assumption, that if methods and instance variables of a class are interconnected, then the class is cohesive. If a method *x* uses instance variable *y*, then *x* and *y* are interconnected. Henderson-Sellers [6] defined the metric LCOM* that is similar to LCOM, but has a fixed scale. Briand et al [2] observed that the scale of LCOM* is from 0 to 2, and gave a refined version of this metric with scale from 0 to 1. Other similar cohesion metrics are LCC and TCC [1], and CBMC [3]. Good surveys of cohesion metrics have been made by Briand et al [2] and Chae et al [3].

A weakness in the cohesion metric research is that it is not clear what kind of problems are the metrics expected to find, and for which kind of programs the metrics give different results. Because of this it is hard to see which are the advantages and disadvantages in using each metric. In addition to this problem, applying the metrics effectively is hard because the cohesion values measured by the metrics have hidden dependencies on other attributes, for example the metrics tend to give lower cohesion for big classes.

Besides contributing results on two variations of cohesion metric for JDK and Eclipse SDK, we also calculated a profile for classes based on how many instance variables their methods use. This will help us to normalize the results so that problematic classes can be found, and properly designed classes will not have to be unnecessarily inspected. In further work, we will study ways to normalize LCOM and evaluate the goodness of such normalization. We also aim to study more variations of LCOM and define some guidelines for good programs, then evaluate a set of programs with these guidelines, and study how the metrics correlate with these guidelines.

2 BASIC LCOM EVALUATIONS

The LCOM value for a class *C* is defined as $\mathrm{LCOM}(C) = 1 - |E(C)| / (|V(C)| \times |M(C)|)$, where $V(C)$ is the set of instance variables, $M(C)$ is the set of instance methods, and $E(C)$ is the set of pairs (v,m) for each instance variable *v* in $V(C)$ that is used by method *m* in $M(C)$. In the following, we consider 'use' to also include *indirect use via other methods*.

The traditional way to measure LCOM is considering only the locally defined variables and methods. We first have to resolve if we should ignore private methods and variables. Then we check for what kind of classes the metric is not applicable at all. For the classes for which the metric is applicable, we need to evaluate if it gives reasonable results.

2.1 Private methods and variables

We compare two LCOM variations LCOM(L,L), including all local instance methods and all local instance variables, and LCOM(L,P), including all local instance variables but only public instance methods. In 15782 of the cases, we have the same value. In most of these cases, there are no private methods, so the values are the same because of that. In 1364 cases we get higher cohesion values by including all methods, and in 2826 cases we get higher values by ignoring the private methods. The classes, where the values are equal, are quite small, and the classes were values differ, are bigger. There are no significant differences between classes where taking all methods into account (LCOM(L,L)) is higher and classes where ignoring private methods (LCOM(L,P)) is higher.

These values also correlate very much, so we should pick another one of them. The alternative with the private methods ignored is more natural, because the non-private methods define the interface and the private methods are just implementation details. Because indirect calculation is used, the private methods can be safely ignored. If the indirect calculation is not used, the situation would be different. Then it would probably be best to include the private methods into the calculation.

There are some cases where the private methods are used by something that is not the defining class. One example are the serialization methods like readObject. Another possibility is using reflection to call the private methods. It might be useful to include methods like readObject or finalize into the computation, but to bypass the class interface by using reflection does not seem a good idea.

Protected methods are also a problem, because they should be used with inheritance, and the local metric does not consider inheritance. We choose not to include the protected methods. The situation with the private variables is similar to the private methods. But now, including the private variables into computation is more natural, since all variables should be defined private anyway.

2.2 Simple classes

There are two reasons for the LCOM(L,P) to be not applicable. First is that a class does not have any locally defined variables, and the second is that a class does not have any locally defined methods. For 7771 classes out of 21036, there are no such variables, and for 4208 classes there are no locally defined public methods. In total, there are 8903 classes without locally defined variables or locally defined methods. The metric LCOM(L,P) is only applicable to less than half of the cases.

In 4208 classes there are no instance methods. Most of these classes (3076) have no instance variables either. These classes include static methods, static variables, and/or inner classes. Another possibility is that a set of classes is defined, and then their dynamic types are used as tags. This is used with for example exceptions.

Then also over third of the classes (7771) have no locally defined instance variables at all. To give an idea, what kind of classes these are, 491 of the names of these classes have suffix Action, 401 Exception, 307 Provider, 186 Factory, 253 Handler, 222 Helper, 207 Listener, 111 Adapter, 132 Util, and 210 Messages.

The suffixes for all measured classes are: 556 Exception, 1148 Action, 558 Provider, 317 Listener, 282 Factory, 237 Adapter, 232 Event, 311 Dialog, 285 Info, 583 Impl, 435 Page, 288 Manager, 249 Helper, 211 Messages, 279 I, 285 Info, and 404 Handler. In 690 classes there is a unique suffix. There are 1582 different suffixes.

The 4695 classes that have methods but no variables, are classes that are used like functions. Similarly, classes that have variables, but no methods, are used as pure data.

3 INTERPRETATION OF MEASURES

In LCOM(L,P) only locally defined variables and public methods are used in calculating the cohesion. This is the traditional way to measure cohesion. However, when

Range	Classes
Trivial	8903
[0.0..0.1)	2865
[0.1..0.2)	447
[0.2..0.3)	541
[0.3..04)	1112
[0.4..05)	740
[0.5..0.6)	1880
[0.6..0.7)	1616
[0.7..0.8)	1250
[0.8..0.9)	1149
[0.9..1.0)	321
[1.0..1.0]	212

Table 1: Distribution of classes according to their LCOM(L,P) value.

calculating variable usage, the indirect computation also considers indirect usage via inheritance.

Table 1 shows that the number of trivial cases (no instance methods) is quite large. The bad cases (1.0 and 0.9..1.0) are interesting as well as the good case (0.0..0.1). In the following, we attempt to classify the reasons behind the observed cohesion values.

3.1 Maximal LCOM(L,P)

Our first question is what kinds of classes have the maximum lack of cohesion. There are 187 such classes. It is odd that a class has instance variables, and then it has instance methods, which do not use these instance variables. In the following, we analyse 5 of them. Most of these classes are defined in Eclipse. An example is org.eclipse.pde.internal.ui.model.plugin.PluginDocumentHandler. This class has protected methods that access the instance variable that is declared, and one public method that uses only inherited variables. In our opinion, this class is well made and the reason for bad LCOM value is that the class is *designed to be used via inheritance*.

The class java.util.AbstractList is similar. It has one instance variable that is used only in an inner class (an iterator) and the idea seems to be that an inheriting class can control the operation of the inner class via that variable. One can also say that in this case the class is *designed to support its inner classes*.

Another example is org.eclipse.jdt.core.dom.Name. It has one variable, index, that is visible to the package, and then there are methods that either use the inherited variables or the this-variable to test the type of the object. In our opinion, this class is not well designed -- one should not expose the instance variable index to the package but to provide methods to manipulate that property. However, perhaps the use of implicit dynamic type variable should be considered as an additional instance variable in the measurements.

The class ComponentViewerPane also has maximal lack of cohesion. It has an instance variable but for using the instance variable, the class provides static methods. In our opinion, there is no good reason to do so.

The Java RMI has class UnicastRemoteObject, which is in a central role in RMI. It declares 3 instance variables but none of its public instance methods use them. Their actual use is related to serialization and reflection. Since this a rather low level system class, it is difficult for us to say whether the bad LCOM(L,P) value in this case is a sign of bad design.

3.2 Low cohesion

The above cases are quite pathological, and the problems that cause them can possibly be detected in some other way. Next we check what kind of classes get very high lack of cohesion, but not the maximal one. Our first observation, based on Table 1, is that these classes tend to be quite big.

The class com.sun.org.apache.xml.internal.serializer.ToXMLSAXHandler has one variable and 37 methods. Its LCOM(L,P) is 0.95, which is due to the fact that most of the methods use its parent class services to implement the required functionality. In this sense, this class is badly implemented, since most of the functionality of this class could be pushed up in the inheritance hierarchy.

The class javax.swing.JTable.AccessibleJTable has 6 variables and 53 methods. Of these, 32 methods do not access any variables, and 11 access only one variable. This class is an inner class, and it uses heavily the attributes of the outer class. The class is *designed to use its outer class* -- making it hard to analyse the class with this metric.

The class org.eclipse.jdi.internal.VirtualMachineImpl has 46 variables and 75 methods. To see, why the cohesion is low, we can check how many connected components the cohesion graph of the class has. If we ignore methods that do not use any variables from the cohesion graph, then there are 15 disjoint components in the class. There are 8 components for singleton patterns. Some components are for variables that have only getters and setters. The huge number of components suggests that this class (or its parent class) is badly designed, but it rather seems that it is *designed to strongly lean on its parent class*.

The class org.eclipse.ui.texteditor.templates.TemplatePreferencePage has 12 variables and 8 methods. There are three disjoint components in the class. One private function is called only from an anonymous inner class and this is not measured. This class is *partly designed to support its inner classes*, but most of the methods are *designed to strongly use its parent class*.

If a method defines an inner class, it might be a good idea to include the instance variables that might be used in that inner class to the set of variables used by the method. Also perhaps non-static inner classes could be counted as methods in the outer class.

The class org.eclipse.jdt.internal.codeassist.InternalCompletionProposal has only one public setter method, the others are protected. The methods are mostly getters and setters. There are 11 variables. This class is *designed to be used via inheritance*. Some protected methods are actually used by other classes in the package, not by the inheriting classes, which is a bit confusing.

The class org.eclipse.pde.internal.ui.model.plugin.PluginDocumentNode has 10 variables and 29 methods. Most methods are getters and setters. In fact, all methods use exactly one variable. This means that the class has 10 disjoint components. The data might still have something in common, but it cannot be seen from this class, rather from classes that are using this class. This class appears to be badly designed.

The class org.eclipse.swt.internal.ole.win32.COMObject has 82 methods and one variable. Of the methods, 80 are stubs that do not use any variables (those return a constant value). Otherwise the class has perfect cohesion. For this kind of classes, ignoring simple methods makes a big difference. This class is *designed to be used via inheritance* (or more likely by mapping native objects into Java).

3.3 Average cohesion

We pick only one example case for cohesion between 0.1 and 0.9. The metric is not very useful here, because so many classes have values that are in this range.

The class org.eclipse.jdt.debug.ui.launchConfigurations.AppletMainTab has 11 variables and 10 methods. LCOM(L,P) is 0.8. The class uses a lot of private methods. Most methods use only few variables, and as is common with classes that have several variables, the LCOM values become high. If methods using no instance variables are ignored, LCOM(L,P) is 0.6. The class has two different components. This split comes from the base class. We cannot conclude that this class is badly designed although the number of variables is rather high -- it is quite ordinary that *all methods do not use all the variables*.

3.4 Minimal LCOM(L,P)

The classes with maximal or almost maximal cohesion are almost always very small, but there are some exceptions. These classes are usually not very interesting, but there are lots of them so we study what they are like.

The class org.eclipse.core.internal.resources.Marker has 2 variables and 20 methods. Of the methods, 17 use both. The class org.eclipse.ui.internal.texteditor.quickdiff.ReferenceSelectionAction has 3 variables and 2 methods. The methods are relatively complex, and there are two private helper methods.

The class org.eclipse.jdt.ui.search.ElementQuerySpecification has only one variable and method. The classes that have maximal cohesion are usually small, as is obvious from the definition. The class com.sun.org.apache.bcel.internal.util.ClassPath.Zip has also just one variable and method.

There are very few big classes that have maximal or almost maximal cohesion. One such class is javax.swing.plaf.synth.ImagePainter. It has 7 variables and 118 methods. Of the methods, 113 methods accesses every variable.

4. CLASS PROFILE INFORMATION FOR LCOM NORMALIZATION

We can decompose LCOM into *profiles*. First we measured how many variables the methods use, see Table 2. Then, to see how the methods use the variables in the classes, we calculated the percentages of methods that use certain number of variables, see Table 3.

Variables used	Local methods	Inherited methods
0	1.83	12.6
1	2.48	10.87
2	0.67	5.81
3	0.35	1.97
4	0.24	1.48
5	0.12	0.72
6	0.1	0.47
7	0.06	0.43
8	0.05	0.19
9	0.04	0.3
10	0.02	0.16
11..	0.11	0.81
All cases	6.1	35.8

Table 2: Variable usage. The table shows the average number of methods that access a certain number of instance variables in the class.

The most obvious difference between inherited and local methods is that there are relatively more inherited methods that do not use any variables, than such methods that are locally defined. This is mostly because the class java.lang.Object has several such methods.

From the Table 3 we can see that there are quite many methods that use all of the variables. Otherwise the average percentages do not depend that much on the number of variables.

The situation is different when inheritance is taken into account. Very small percentage uses all of the variables, including inherited ones. There is also another kind of profile that can be calculated. In it we measure for variables, in how many methods they are accessed.

	0	1	2	3	4	5	6	7	8	9	10+	LCOM	Classes
1	22	78										0.22	3589
2	15	46	39									0.38	2351
3	13	44	16	27								0.48	1579
4	12	42	14	11	21							0.53	1150
5	12	40	13	11	9	15						0.58	786
6	9	45	12	7	7	7	13					0.62	550
7	11	45	11	7	6	5	5	21				0.66	427
8	11	37	11	7	6	5	4	5	13			0.63	319

9	13	42	11	7	4	5	4	4	4	6		0.71	235
10	10	40	10	7	4	6	5	4	3	4	6	0.7	188
11..20	10	43	10	6	4	4	4	3	2	2	13	0.73	726
20..	12	39	11	7	4	3	3	3	2	2	15	0.79	232
All	15	54	14	7	4	2	1	1	0	0	1	0.44	12132

Table 3: The first column separates classes according to how many local instance variables those define. The next 11 columns list the average percentage of classes using the amount of variables listed in the first row.

5 CONCLUSIONS

Good LCOM(L,P) value for a class indicates that the class is well designed, but a bad value does not always indicate bad design. A very good value typically indicates that the class has only a few instance variables. Because usually in inheritance, the extension is orthogonal or almost orthogonal, LCOM(L,P) usually gives a good measure of the cohesion of the class.

Table 3 clearly indicates that it is characteristic for LCOM that bigger classes get low and smaller classes get high cohesion values. For quite many classes the LCOM(L,P) value is bad although there is only one component in the sense of LCOM(L,P), because the class has rather many instance variables and the methods typically use only a few of them. We find that in most cases this is not a sign of bad design. The methods of a class can be seen to form a distribution with respect to the number of instance variables those advance. In our opinion, it is useful to normalize LCOM(L,P) with respect to such average distribution. This is especially meaningful for bigger classes, since those typically have a lot of methods using only a small number of variables. An exact formulation for the normalization we leave as future work.

Very bad values also have other explanations. We found that, besides bad design, the reason in such cases can be

- Designed to be used via inheritance,
- Mainly used inherited features to implement own features,
- Designed to support inner classes, and
- Desgined to advance outer class.

The LCOM(L,P) should not be used for those cases.

REFERENCES

[1] Bieman J.M., and B.-K. Kang. Cohesion and reuse in an object-oriented system. In SSR'95: Proceedings of the 1995 Symposium on Software reusability, pages 259--262, New York, NY, USA, 1995. ACM Press.

[2]Briand L.C., J.W. Daly, and J. Wust. A unified framework for cohesion measurement in object-oriented systems. Empirical Software Engineering: An International Journal, 3(1):65--117, 1998.

[3] Chae H.S., Y.R. Kwon, and D.H. Bae. A Cohesion Measure for Object-Oriented Classes. Software - Practice & Experience, 30(12):1405--1431, 2000.

[4] Chidamber S.R. and C.K. Kemerer. Towards a Metric Suite for Object Oriented Design. In Proceedings, OOPSLA'91, Sigplan Notices 26(11), pages 197 -- 211, 1991.

[5] Chidamber S.R. and C.K. Kemerer. A Metrics Suite for Object Oriented Design. IEEE Transactions on Software Engineering, 20(6), June 1994.

[6] Henderson-Sellers B. Object-Oriented Metrics: Measures of Complexity. Prentice Hall, 1996.

ABOUT THE AUTHORS

PhD student Sami Mäkelä and PhD Ville Leppänen, Department of Information Technology, University of Turku, Finland, E-mail: {Sami.Makela,Ville.Leppanen}@it.utu.fi.