Run-Time Cohesion Metrics: An Empirical Investigation

Áine Mitchell*
Department of Computer Science
National University of Ireland, Maynooth,
Co. Kildare, Ireland
ainem@cs.may.ie

James F. Power
Department of Computer Science
National University of Ireland, Maynooth,
Co. Kildare, Ireland
jpower@cs.may.ie

Abstract

Cohesion is one of the fundamental measures of the 'goodness' of a software design. The most accepted and widely studied object-oriented cohesion metric is Chidamber and Kemerer's Lack of Cohesion in Methods measure. However due to the nature of object-oriented programs, static design metrics fail to quantify all the underlying dimensions of cohesion, as program behaviour is a function of it operational environment as well as the complexity of the source code. For these reasons two run-time object-oriented cohesion metrics are described in this paper, and applied to Java programs from the SPECjvm98 benchmark suite. A statistical analysis is conducted to assess the fundamental properties of the measures and investigate whether they are redundant with respect to the static cohesion metric. Results to date indicate that run-time cohesion metrics can provide an interesting and informative qualitative analysis of a program and complement existing static cohesion metrics.

1. Introduction

The most accepted and widely studied object-oriented cohesion metric is Chidamber and Kemerer's Lack of Cohesion in Methods measure [8]. This is considered to be the seminal cohesion metric for methods in a class, and is designed to give a qualitative measure of the *internal complexity* of a software design. Attempts have been made to improve upon this measure to better encapsulate the cohesiveness of a class, but despite the number of theoretical and empirical studies conducted in this area, there is still no generally accepted definition or measure of object-oriented cohesion [4, 5, 6, 7].

Many of the shortcomings of Chidamber and Kemerer's measure can be inferred from its definition. Suppose a class contains n methods, $m_1, \ldots, m_n$, and let $\{I_i\}$ be the set of instance variables referenced by method m_i. We can define two disjoint sets:

$$\begin{array}{rcl} P & = & \{(I_i, I_j) \mid I_i \cap I_j = \emptyset\} \\ Q & = & \{(I_i, I_j) \mid I_i \cap I_j \neq \emptyset\} \end{array} \quad (1)$$

The lack of cohesion in methods is then defined from the cardinality of these sets, by:

$$S_{LCOM} = \begin{cases} |P| - |Q| & \text{if } |P| > |Q| \\ 0 & \text{otherwise} \end{cases} \quad (2)$$

We refer to this from now on as *static* LCOM, S_{LCOM}, to emphasise that it is calculated based on a static analysis of the source code, in contrast to the *run-time* metrics defined below.

A previous study has shown that using this definition a large number of classes get assigned a zero value for S_{LCOM}[1]. If there is only a single method in a class accessing instance variables in the same class, only one set is formed for that class and therefore a zero value is assigned. Also if $|P| < |Q|$, the number of pairs of methods having no common instance variables is less than the number of pairs of methods having common instance variables, the metric is again set to zero. Studies have shown that this metric sets cohesion to zero for classes that were intuitively judged to have very different cohesions [5, 6].

Another inadequacy with this metric is due to the fact that it designed for use at the *design* stage of the software life-cycle. Object-oriented software tends to evolve and become more complex over time. The use of legacy code in object-oriented systems may lead to code becoming 'dead' or 'obsolete' and this may adversely affect the ability of static metrics to provide an accurate measure of cohesion. In addition, some data affecting cohesion metrics can only be calculated from run-time information. Features of object-oriented programming such as polymorphism, dynamic binding and inheritance render the static cohesion metrics imprecise as they do not reflect perfectly the run-time situation. The above reasons led us to investigate whether having a *run-time* cohesion measure would result in an improvement over the S_{LCOM}.

*Please address correspondence to Áine Mitchell at the above address

Accepted for the International Conference on Software Engineering Research and Practice (SERP '04)
Las Vegas, Nevada, USA, June 21-24, 2004, pp. 532-537

1.1. Related Work

A similarly-motivated study was conducted by Gupta and Rao [12] which measured *module* cohesion in legacy software. Gupta and Rao compared statically calculated metrics against a program execution based approach of measuring the levels of module cohesion. The results from this study showed that the static approach significantly overestimated the levels of cohesion present in the software tested. However, Gupta and Rao were considering C programs, where many features of object-oriented programs are not directly applicable.

A number of studies of the dynamic behaviour of Java programs have been carried out, mostly for optimisation purposes. Issues such as bytecode usage [11] and memory utilisation [9] have been studied, along with a comprehensive set of "dynamic metrics" relating to polymorphism, object creation and hot-spots [10]. However, none of this work directly addresses the calculation of standard software metrics at run-time.

In previous work we have formulated run-time definitions of cohesion metrics [14], and conducted an informal exploration of their properties. Section 2 briefly summarises our definition of these metrics and Section 3 describes their implementation. The principal contribution of this paper is the use of statistical analysis to determine the usefulness or redundancy of these metrics. Sections 4, 5 and 6 follows [3] in analysing the metrics, and Section 7 discusses the results obtained.

2. Run-time cohesion metrics

The first run-time metric we define is **Run-time Simple LCOM** (R_{LCOM}). This is a direct extension of the static case, except that now we only count instance variables that are actually accessed at run-time. Thus, for a set of methods $m_1,\ldots,m_n$, as before, let $\{I_i^R\}$ represent the set of instance variables referenced by method m_i at run-time. We can define P^R and Q^R by substituting I_i^R for I_i in equation 1 above

$$\begin{aligned} P^R &= \{(I_i^R, I_j^R) \mid I_i^R \cap I_j^R = \emptyset\} \\ Q^R &= \{(I_i^R, I_j^R) \mid I_i^R \cap I_j^R \neq \emptyset\} \end{aligned} \tag{3}$$

We can then define R_{LCOM} as:

$$R_{LCOM} = \begin{cases} |P^R| - |Q^R| & \text{if } |P^R| > |Q^R| \\ 0 & \text{otherwise} \end{cases} \tag{4}$$

We note that for any method m_i, $(I_i - I_i^R) \geq 0$, and represents the number of instance variables mentioned in a method's code, but not actually accessed at run-time. As a consequence of this, it is always the case that $R_{LCOM} \leq S_{LCOM}$.

It is reasonable to suggest that a heavily accessed variable should make a greater contribution to class cohesion that one which is rarely accessed. However, the (R_{LCOM}) does not distinguish between the degree of access to instance variables. Thus we define a second run-time measure **Run-time Call-Weighted LCOM** (RW_{LCOM}) by weighting each instance variable by the number of times it is accessed at run-time.

As before, consider a class with n methods, $m_1,\ldots,m_n$, and let $\{I_i\}$ be the set of instance variables referenced by method m_i. Define N_i as the number of times method m_i dynamically accesses instance variables from the set $\{I_i\}$.

Now define a *call-weighted* version of equation 1 by summing over the number of accesses:

$$\begin{aligned} P^W &= \sum_{1 \leq i,j \leq n} \{(N_i + N_j) \mid I_i \cap I_j = \phi\} \\ Q^W &= \sum_{1 \leq i,j \leq n} \{(N_i + N_j) \mid I_i \cap I_j \neq \phi\} \end{aligned} \tag{5}$$

$$\text{where} \qquad P^W = 0, \text{if} \{I_1\}, ..., \{I_n\} = \phi$$

Following equation 2 and 4 we define:

$$R_{LCOM} = \begin{cases} |P^R| - |Q^R| & \text{if } |P^R| > |Q^R| \\ 0 & \text{otherwise} \end{cases} \tag{6}$$

3. Experimental Platform

The Java Virtual Machine (JVM) provides an ideal environment within which to study the operation of Java programs. In particular, Sun Microsystem's Java 2 SDK version 1.4.0_0 provides a multi-tiered debugging architecture called the Java Platform Debug Architecture (JPDA) [18]. The JPDA provides introspective access to a running JVM's state including the class, array, interface, and primitive types, and instances of those types.

In order to match objects against method calls it is necessary to model the execution stack of the JVM, as this information is not provided directly by the JPDA. We have implemented an `EventTrace` analyser class in Java, which carries out a stack based simulation of the entire execution in order to obtain information about the state of the execution stack. This class also implements a filter which allows the user to specify which events and which of their corresponding fields are to be captured for processing. This allows a high degree of flexibility in the collection of the dynamic trace data.

The final component of our collection system is a `Metrics` class, which is responsible for calculating the desired metrics on the fly. It is also responsible for outputting the results in text format. The metrics to be calculated can be specified from the command line. The addition

Accepted for the International Conference on Software Engineering Research and Practice (SERP '04)
Las Vegas, Nevada, USA, June 21-24, 2004, pp. 532-537

Table 1. Descriptive Statistic Test Results for all Programs

_201_compress	Mean	SD
R_{LCOM}	2.18	6.48
RW_{LCOM}	452.50	1624.11
S_{LCOM}	33.59	118.25

_202_jess	Mean	SD
R_{LCOM}	0.95	3.56
RW_{LCOM}	4566543.45	6563246.65
S_{LCOM}	10.06	64.84

_205_raytrace	Mean	SD
R_{LCOM}	2.78	7.54
RW_{LCOM}	543222.56	644337.77
S_{LCOM}	25.21	96.45

_209_db	Mean	SD
R_{LCOM}	5.43	9.99
RW_{LCOM}	344888.79	789838.96
S_{LCOM}	52.86	144.75

_213_javac	Mean	SD
R_{LCOM}	0.655	6.43
RW_{LCOM}	645563.53	852543.65
S_{LCOM}	7.517	48.57

_222_mpegaudio	Mean	SD
R_{LCOM}	1.75	4.553
RW_{LCOM}	3009148.02	12924046.56
S_{LCOM}	18.35	82.64

_227_mtrt	Mean	SD
R_{LCOM}	2.82	7.63
RW_{LCOM}	543355.56	644853.77
S_{LCOM}	25.21	96.45

_228_jack	Mean	SD
R_{LCOM}	2.15	7.07
RW_{LCOM}	182828.675	622422.79
S_{LCOM}	19.95	89.44

of the metrics class allows new metrics to be easily defined as the user need only interface with this class. See [15, 16] for additional information.

To investigate the robustness of our metrics we carried out measurements on the Java programs from the SPECjvm98 benchmark suite [17]. In the following sections the data collected using the above technique is analysed to determine if the run-time cohesion metrics are redundant with respect to the static LCOM measure. The procedure is derived from the method outlined in [3], and consists of descriptive statistics, a correlation study and principal component analysis.

4. Descriptive Statistics

For each program in the SPECjvm98 suite the **distribution** (mean) and **variance** (standard deviation) of each measure across the classes is calculated. These statistics are used to select metrics that exhibit enough variance to merit further analysis, as a low variance metric would not differentiate classes very well and therefore would not be a useful predictor of external quality. Descriptive statistics will also aid in explaining the results of the subsequent analysis.

The descriptive statistics for the SPECjvm98 benchmark programs are illustrated by Table 1. The measures were shown to exhibit large variances which makes them suitable candidates for further analysis.

The subsequent statistical techniques all require a **normal (bivariate) data distribution**. The Shapiro-Wilk test was used to test whether the data was normally distributed. Any data that did not exhibit a normal distribution was transformed by calculating the logarithm of each data point.

5. Correlation Study

A **correlation study** was undertaken to investigate how strongly the metrics are related. The Pearson or product moment correlation test was used. The correlation coefficient *r* is a number that summarises the direction and degree of linear relations between two variables and is also known as the Pearson Product-Moment Correlation Coefficient. *r* can take values between -1 through 0 to +1. When the correlation is positive ($r > 0$), as the value of one variable increases, so does the other. The closer *r* is to zero the weaker the relationship. If a correlation is negative, when one variable increases, the other variable decreases.

The following general categories indicate a quick way of interpreting a calculated *r* value:

0.0 to 0.2 Very weak to negligible correlation
0.2 to 0.4 Weak, low correlation (not very significant)
0.4 to 0.6 Moderate correlation
0.7 to 0.9 Strong, high correlation
0.9 to 1.0 Very strong correlation

Any relationship between two variables should be assessed for its *significance* as well as its strength. A standard two tailed t-test was used to determine whether the correlation coefficient was statistically significant. Coefficients were considered significant if the t-test p-value was below 0.05. This tells how unlikely a given correlation coefficient, *r*, will occur given no relationship in the population. Therefore the smaller the p-level, the more significant the relationship.

The results for the Pearson correlation coefficient test for the programs under evaluation are illustrated by Table 2. As R_{LCOM} is essentially S_{LCOM} compounded at run-time some degree of correlation between these two would be expected. However, while some cases exhibited a weak de-

Table 2. Pearson Correlation Coefficient Test Results for all Programs, with a p-level of 0.05

Program	S_{LCOM} *vs.* R_{LCOM}	S_{LCOM} *vs.* RW_{LCOM}	R_{LCOM} *vs.* RW_{LCOM}
_201_compress	0.455	0.553	0.984
_202_jess	0.242	0.143	0.765
_205_raytrace	0.434	0.343	0.675
_209_db	0.424	0.516	0.892
_213_javac	0.563	0.643	0.732
_222_mpegaudio	0.173	0.054	0.721
_227_mtrt	0.444	0.348	0.681
_228_jack	0.386	0.465	0.515

gree of correlation, none of the programs yielded a significant correlation.

In contrast, the rightmost column of Table 2 demonstrates a significant correlation between R_{LCOM} and RW_{LCOM}. Five of the programs exhibit a strong correlation, with the remaining three exhibiting a moderate correlation. Thus, there is a strong indication that R_{LCOM} and RW_{LCOM}, despite the difference in their definition, may not be independent measures of run-time class cohesiveness.

6. Principal Component Analysis

Principal Component Analysis (PCA) is used to analyse the covariate structure of the metrics and to determine the underlying structural dimensions they capture. In other words PCA can tell if all the metrics are likely to be measuring the same class property. The number of principal components will be decided based on the amount of variance explained by each component. A typical threshold would be retaining principal components with eigenvalues (variances) larger than 1.0. This is the Kaiser criterion [13].

Table 3 shows the results of the principal component analysis when all of the metrics are taken into consideration. Using the Kaiser criterion to select the number of factors to retain we find that the metrics mostly capture two orthogonal dimensions in the sample space formed by all measures. In other words for each of the programs analysed two principal components are retained.

The R_{LCOM} metric belongs to the same principal component as the RW_{LCOM} in all cases. As the Pearson correlation coefficient test showed that these two measures are also strongly correlated, it may be sufficient to evaluate the R_{LCOM} metric alone. In other words not enough variance is captured by the RW_{LCOM} measure that is not accounted for by R_{LCOM}.

However, a significant amount of variance is captured by the run-time metrics that is not accounted for by S_{LCOM} as S_{LCOM} was found to be weakly correlated or not at all with the run-time measures. Analysing the definitions of the measures that exhibit high loadings in PC1 and PC2 yields the following interpretation of the cohesion dimensions:

PC1 Measures R_{LCOM} and RW_{LCOM}

PC2 Measures S_{LCOM}

In summary, the finding from this study indicate that no significant information about the cohesiveness of a class can be gained by evaluating the RW_{LCOM} instead of the simpler R_{LCOM}. However, the PCA results seem to suggest that R_{LCOM} is not redundant with respect to S_{LCOM} and that it captures additional information about cohesion. The values show that R_{LCOM} is not just a surrogate static measure. Clearly the simple static calculation of LCOM masks a considerable amount of detail available at run-time.

7. Discussion

Besides the lack of discriminating power of these cohesion (static and run-time) metrics there are a number of other inadequacies that should be addressed.

7.1. Inclusion of Access Methods

The role of an access method is typically to provide read or write access to an instance variable belonging to a class. Usually these methods will deal with a single instance variable. The result of this may be the presence of many pairs of such methods that do not have any instance variables in common. This is a disadvantage when calculating metrics which involve counting pairs of methods that use common instance variables. Their presence may artificially decrease the value of the cohesion measure.

The cohesion value may also be artificially decreased by the presence of access methods if other methods of the class use the access method to access the instance variable instead of directly referencing it. This will result in a reduction in the number of references to that instance variable.

7.2. Inclusion of Constructors

The role of a constructor is to assign initial values to the instance variables of a class. Typically most of the instance variables in a class will be accessed by these constructor methods. The inclusion of such methods in the analysis will result in an artificial increase in cohesion, as many pairs of methods will exist that have instance variables in common.

7.3. Impact of Inheritance

The metrics used in this study make no attempt to deal with methods and instance variables that may be inherited

Table 3. Principal Component Analysis Test Results for all Programs

_201_compress	PC1	PC2
R_{LCOM}	**0.979**	0.192
RW_{LCOM}	**0.948**	0.313
S_{LCOM}	0.488	**0.615**

_202_jess	PC1	PC2
R_{LCOM}	**0.786**	0.213
RW_{LCOM}	**0.845**	0.132
S_{LCOM}	0.464	**0.766**

_205_raytrace	PC1	PC2
R_{LCOM}	**0.845**	0.145
RW_{LCOM}	**0.823**	0.089
S_{LCOM}	0.145	**0.535**

_209_db	PC1	PC2
R_{LCOM}	**0.846**	0.108
RW_{LCOM}	**0.903**	0.046
S_{LCOM}	0.499	0.421

_213_javac	PC1	PC2
R_{LCOM}	**0.566**	0.323
RW_{LCOM}	**0.643**	0.034
S_{LCOM}	0.487	0.467

_222_mpegaudio	PC1	PC2
R_{LCOM}	**0.813**	0.006
RW_{LCOM}	**0.807**	0.013
S_{LCOM}	0.001	**0.992**

_227_mtrt	PC1	PC2
R_{LCOM}	**0.843**	0.143
RW_{LCOM}	**0.821**	0.087
S_{LCOM}	0.143	**0.534**

_228_jack	PC1	PC2
R_{LCOM}	**0.714**	0.280
RW_{LCOM}	**0.690**	0.002
S_{LCOM}	0.314	**0.541**

by a class. This approach is deemed cohesion at the class level. It deals with the relationships between the elements of a class, that is, all of its non-inherited methods and instance variables.

A number of alternatives have been proposed to deal with such inheritance issues. Bieman and Kang [2] proposed to include inherited instance variables but exclude inherited methods. Briand et al [4] suggested to include inherited methods but exclude inherited instance variables. However, neither party provided a convincing argument why either of these approaches would yield a more accurate result.

7.4. Distinction Between Directly and Indirectly Connected Pairs of Methods

The distinction between directly and indirectly connected pairs of methods is not addressed. If two distinct methods m_1 and m_2 of a class both access an instance variable of that class, they are said to be *similar methods*. They are also deemed to be directly connected to one another. If another method m_3 in the class is similar to method m_2 it is said to be *directly* connected to m_2 but *indirectly* connected to m_1.

These metrics count direct connections only. The disadvantage of this is that it has been proposed that indirect connections appear to give a better indication for when to break up a class [4]. For a full review of these and other possible inadequacies with the current run-time definitions see [14].

8. Conclusion and Future Work

This paper reviewed two run-time cohesion metrics designed to quantify the external quality of an object-oriented application. A method for collecting such measures was described which utilised the Java Platform Debug Architecture. An empirical investigation of the metrics was conducted using programs from the SPECjvm98 benchmark suite.

The differences in the underlying dimensions of cohesion captured by static versus run-time cohesion metrics was assessed using a correlation study and principal component analysis. The investigation was conducted using the static LCOM metric (S_{LCOM}) as defined by Chidamber and Kemerer. The results indicated that the R_{LCOM} metric captured additional information about cohesion than S_{LCOM} alone. Results indicate that it is worthwhile to continue the investigation into run-time cohesion metrics and their relationship with the external quality.

There are plans to extend this work in a number of ways. We hope to develop a comprehensive set of run-time object-oriented metrics that can intuitively quantify such aspects of object-oriented applications such as inheritance, dynamic binding, polymorphism and dynamic binding. It is also a goal to improve on the additional performance overhead that results from the use of the JPDA during the collection of the dynamic trace information.

9. Acknowledgements

This work is funded by the Embark initiative, operated by the Irish Research Council for Science, Engineering and Technology (IRCSET).

References

[1] V.R. Basili, L.C. Briand, and W.L. Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on Software Engineering*, 22(10):751–761, October 1996.

[2] J.M. Bieman and B.K. Kang. Cohesion and reuse in an object-oriented system. In *Proc. ACM Symp. Software Reusability (SSR'94)*, pages 295–262, 1995.

[3] L.C. Briand. Empirical investigations of quality factors in object-oriented software. In *Empirical Studies of Software Engineering*, Ottawa, Canada, March 4–5 1999.

[4] L.C. Briand, J.W. Daly, and J.K. Wüst. A unified framework for cohesion measurement in object-oriented systems. *Empirical Software Enggineering: An Int'l J.*, 3(1):65–117, 1998.

[5] L.C. Briand, S. Morasca, and V. Basili. Defining and validating high-level design metrics. Technical Report CS-TR 3301, Department of Computer Science, University of Maryland, College Park, MD 20742, USA, 1994.

[6] L.C. Briand, J.K. Wüst, J.W. Daly, and V. Porter. Exploring the relationship between design measures and software quality in object-oriented systems. *The Journal of Systems and Software*, 51:245–273, 2000.

[7] D.N. Card, V.E. Church, and W.W. Agresti. An empirical study of software design practices. *IEEE Transactions on Software Engineering*, 12(2):264–271, 1986.

[8] S.R. Chidamber and C.F. Kemerer. A metrics suite for object-oriented design. *IEEE Transactions on Software Engineering*, 20(6):467–493, June 1994.

[9] Jong-Deok Choi, Manish Gupta, Mauricio J. Serrano, Vugranam C. Sreedhar, and Samuel P. Midkiff. Stack allocation and synchronization optimizations for java using escape analysis. *ACM Transactions on Programming Languages and Systems*, 25(6):876 – 910, November 2003.

[10] Bruno Dufour, Karel Driesen, Laurie J. Hendren, and Clark Verbrugge. Dynamic metrics for Java. In *Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 149–168, Anaheim, CA, USA, October 26-30 2003.

[11] David Gregg, James Power, and John Waldron. Platform independent dynamic java virtual machine analysis: the java grande forum benchmark suite. *Concurrency and Computation: Practice and Experience*, 15(3-5):459–484, March 2003.

[12] N. Gupta and P. Rao. Program execution based module cohesion measurement. In *16th International Conference on Automated on Software Engineering (ASE '01)*, San Diego, USA, Nov 2001.

[13] I.T. Jolliffe. *Principal Component Analysis*. Springer Verlag, 2nd edition, 2002.

[14] Á. Mitchell and J.F. Power. Run-time cohesion metrics for the analysis of Java programs - preliminary results from the SPEC and Grande suites. Technical Report NUIM-CS-TR2003-08, Department of Computer Science, N.U.I. Maynooth, Co. Kildare, Ireland, April 2003.

[15] Á. Mitchell and J.F. Power. Toward a definition of run-time object-oriented metrics. In *Proceedings of the 7th ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering (QAOOSE'2003)*, Darmstadt, Germany, July 22 2003.

[16] Á. Mitchell and J.F. Power. An approach to quantifying the run-time behaviour of Java GUI applications. In *Winter International Symposium on Information and Communication Technologies*, Cancun, Mexico, Jan 5–8 2004.

[17] SPEC. SPEC releases SPEC JVM98, first industry-standard benchmark for measuring Java virtual machine performance. Press Release, August 19 1998. http://www.specbench.org/osg/jvm98/press.html.

[18] Sun Microsystems, Inc. Java platform debug architecture (JPDA). http://java.sun.com/products/jpda.

Accepted for the International Conference on Software Engineering Research and Practice (SERP '04)
Las Vegas, Nevada, USA, June 21-24, 2004, pp. 532-537