

Colorado Technical University



**Technical Report
Computer Science**

Ratio scale cohesion and coupling metrics for object oriented designs

Konrad Schmidt
konrad@att.net

**Technical Report Number
CTU-CS-2002-007**

Ratio scale cohesion and coupling metrics for object oriented designs

Konrad Schmidt

ABSTRACT

Cohesion, the “relatedness” of module components, and coupling, the “interconnections” between program units, are widely believed to be good indicators of program fault-proneness. In particular, it is believed that programs displaying high cohesion and low coupling are less fault-prone and more maintainable than programs of lower cohesion and/or higher coupling. The premier work in the area of metrics for object oriented design is Chidamber and Kemerer’s 1994 paper “A Metrics Suite for Object Oriented Design”, which proposes, among others, a Lack of Cohesion in Methods (LCOM) metric and a Coupling Between Object Classes (CBO) metric. While the LCOM and CBO metrics are an important first step, they do have some serious limitations. LCOM does not measure cohesion directly, but rather measures the *lack* of cohesion, which is defined to be the number of disjoint sets formed by the intersection of the common instance variables used by an object’s methods. Therefore, if the question is “how hot is it”, LCOM can only say it is “not too cold.” CBO is a simple count of the couplings of one object to another, without regard to the complexity or propriety of a given coupling. In addition, the metric only applies to a given object, and does not give an indication of overall system coupling. Both metrics are simple (interval scale) counts, and cannot be used to measure the relative “goodness” of one object or system to another. It is only possible to say that class A has, say, more couplings than class B. This paper proposes ratio scale metrics for cohesion and coupling, and describes a method whereby the proposed metrics may be validated. The proposed metrics enable one to say that system A, for example, displays better cohesion (or looser coupling) than system B. In addition, the paper postulates that, contrary to popular belief, cohesion is *not* a valid measure of fault-proneness in object oriented designs.

INTRODUCTION

Cohesion refers to the “relatedness of module components.”[2] That is, in an object-oriented design, the methods and instance variables of a given object should be exactly what

is required for the module to perform its function – nothing more, nothing less. Another term for cohesion is *specialization*. It is desirable for each object to be highly specialized in whatever that object’s purpose is, and to know as little as possible about the specialty of any other object. The more specialized an object is, the higher that object’s cohesion.

Coupling is an indication of the strength of the interconnections between program units.[12] Another term for coupling is *independence*, and it is desirable for each object to be as independent as possible from other objects. Highly independent objects are loosely coupled, while highly dependent objects are tightly coupled.

Software developers strive for systems that are highly cohesive and loosely coupled - that is, systems built on objects that are highly specialized and highly independent - in the belief that this represents a *good* design. The definition of a *good* design will vary with the objectives of the designer, and may include many, often contradictory, goals such as producing the most efficient code possible, minimizing the size of the implementation, maximizing maintainability, or minimizing the number of program errors.

This paper presents measures/metrics for coupling and cohesion in an object-oriented design, and proposes a method whereby the metrics can be validated.

Cohesion

While it is widely believed that cohesion is an important design element[3, 4, 11-13], it is also very difficult to measure[10]. Chidamber and Kemerer suggest a *lack* of cohesion metric[5] that uses an object’s instance variables and methods to provide an inverse measure of the similarity of the methods in an object. In their original 1991 work[5] they begin with a “degree of similarity” metric which is simply the count of instance variables used by *all* of an object’s methods. If there is no instance variable referenced by each and every method, the object’s degree of similarity is zero. Unfortunately, the metric cannot distinguish between the case where each method operates on a unique set of instance variables, and where only one method operates on a unique set of instance variables.

Example:

If I_j is the set of instance variables referenced by method M_j , then the object has n such sets, where n is the total number of methods in the object. For $n=3$, the following are the set of instance variables for two different objects: $\{[a,b,c],[d,e,f],[g,h,i]\}$, $\{[a,b,c],[a,b,c],[d]\}$.

The degree of similarity is computed as $\{I_1\} \cap \{I_2\} \dots \cap \{I_n\}$. For each of the objects, the computed degree of similarity is zero, even though the second example appears to display much more cohesion than the first.

For this reason, Chidamber and Kemerer introduced the Lack of Cohesion in Methods (LCOM) metric and defined it to be the number of disjoint sets formed by the intersection of the n sets. In the above example, the LCOM is 3 for the first object, and 1 for the second object, which indicates that the first object displays less cohesion than the second. In their subsequent 1994 paper[6], and for unknown reasons, Chidamber and Kemerer refined the calculation of LCOM as follows:

Let $P = \{(I_i, I_j) | I_i \cap I_j = \emptyset\}$. That is, P is the count of instance sets with no common elements. In the previous example, the first set of I has no common instance variables, so P is three. In the second object, the intersection of I_1 and I_2 is nonempty, the intersection of I_1 and I_3 is empty, and the intersection of I_2 and I_3 is empty. Therefore, P is 2 for the second object.

Let $Q = \{(I_i, I_j) | I_i \cap I_j \neq \emptyset\}$. That is, Q is the count of instance sets with at least one common element. In the previous example, the first set of I has no common instance variables, so Q is zero. In the second object, the intersection of I_1 and I_2 is nonempty, the intersection of I_1 and I_3 is empty, and the intersection of I_2 and I_3 is empty. Therefore, Q is 2 for the second object.

$LCOM = |P| - |Q|$, if $|P| > |Q|$; 0 otherwise. That is, LCOM is the count of instance sets with no common elements minus the count of instance sets with at least one common element, but not less than zero.

Proposed Cohesion Metric

Despite some rather convincing arguments that cohesion is an insignificant factor in object-oriented designs[1], the employer nonetheless wishes to measure it. As previously discussed, Chidamber and Kemerer proposed a Lack of Cohesion metric (LCOM) to provide a measure for the inverse of cohesion. For our purposes, there are two serious shortcomings in this metric: the metric is, at best, an interval scale (but more likely an ordinal scale); it provides an inverse measure that gives no feel for how cohesive an object is. As an analogy, if the question is “how hot is it”, this metric can only tell us that it is “not too cold.” It would be beneficial to have a direct measure of the cohesion of an object, and to have that measure

be on a ratio scale. The new metric uses an object's methods and instance variables as the building blocks, and yields an overall Cohesion Index in the range $\{0..1\}$ for each method, for the object as a whole, and for the overall system.

Let IV_c be the number of common instance variables for a given class c , where common instance variables are those instance variables whose scope is visible to all methods in a class. Under this definition, private variables (variables private to a given method) are not common instance variables. No distinction is made between the various types of variables. That is, an array is treated the same as a single integer, and an integer is indistinguishable from a real number or character string.

Given that there are M methods in the class ($M > 0$), and S classes in the overall system ($S > 0$), let I_j be the number of instance variables referenced by method j , where j is in the range $\{1..M\}$.

As some researchers[8] have suggested that software metrics are meaningless unless size is taken into account, let $SLOC_m$ be the total number of source lines of code in method m , and $SLOC_c$ be the total number of source lines of code in a given class c , such that $SLOC_c =$

$$\sum_{m=1}^M SLOC_m .$$

The cohesion factor for a given method m is then given as: $CM_m = \frac{I_m}{IV_c} * \frac{SLOC_m}{SLOC_c}$, if $IV_c > 0$, and 0 otherwise. This hypothesizes that the cohesion of a given method is related directly to the number of common instance variables referenced by that method. A method that references all of the common instance variables has a cohesion factor of 1.0 (weighted by the relative size of the given method), while a method that references no common instance variables has a cohesion factor of 0.0 (irrespective of the method's relative size).

The cohesion factor for class c as a whole is expressed as: $CO_c = \frac{\sum_{j=1}^M CM_j}{M} * \frac{SLOC_c}{\sum_{k=1}^S SLOC_k}$.

This hypothesizes that the cohesion for the class is the arithmetic mean of the cohesion factors of the methods in the class weighted by the relative size of the class.

The cohesion factor for the overall system is then computed as:

$$CS = \frac{\sum_{c=1}^s CO_c}{s}, \text{ or the arithmetic mean of the cohesion factors of the constituent classes.}$$

Coupling

Coupling is a measure of the strength of the interconnections between program units. Tightly coupled systems have strong interconnections, with object classes highly dependent on each other, while object classes of loosely coupled systems are more independent.

Chidamber and Kemerer define a *Coupling Between Object Classes* (CBO) metric[5] that uses the interaction of a system's objects to provide a measure of the coupling within the system. One object is coupled to another if one of the objects uses methods or instance variables of the other. The CBO for an object is the count of other objects to which the object is coupled. While some inter-object coupling is necessary, excessive or inappropriate coupling is detrimental to modular design and could decrease maintainability.[5] Coupling is not associative. That is, $A \Rightarrow B$ and $B \Rightarrow C$ does not imply $A \Rightarrow C$.

Proposed Coupling Metric

Chidamber and Kemerer proposed a Coupling Between Object Classes (CBO) metric, which is a count of the couplings of one object to another. This provides a simple interval scale measure of individual objects, but says nothing of the overall system coupling. It would be beneficial to have a ratio scale coupling measure for individual objects, as well as for the overall system. In addition, CBO treats coupling as a binary relation – an object is either coupled to another object or it isn't. The proposed coupling metric recognizes that not all couplings are equal[11]. For example, access to instance variables constitutes stronger coupling than simple message passing, and messages with wide parameter lists are more tightly coupled than messages with narrower parameter lists. The proposed metric takes the following factors into consideration:

- Direct access to an instance variable in another object is generally a bad idea, and is regarded as very tight coupling.
- Message passing is regarded as loose coupling, but the coupling tightens with:

- The type of the parameter – simple data types represent looser coupling than complex data types defined within one of the objects.
- Increasing number of parameters in the message.
- A coupling factor should be computed for each individual class, and for the overall system.
- To be consistent with the cohesion factor, the coupling factor should be zero for very tightly coupled classes, and 1 for very loosely coupled classes. As a result, for both newly proposed metrics, the factor is larger for the more desirable case, and smaller for the less desirable case. In this sense, the coupling factor would probably be more accurately described as a *lack of coupling* factor, since a large (close to 1.0) value represents a loosely coupled system. Nonetheless, the term “coupling factor” will continue to be used in this paper.

In the spirit of COCOMO II, which uses *cost drivers* in the estimation of software development effort[7], the following Coupling Component Costs (CCC's) are defined:

Component	Cost
Direct access to instance variable of another class	100
Simple data type parameter	1
Compound data type – defined outside of class	10
Compound data type – defined within class	100

The coupling factor for a given class c can then be expressed as: $CF_c = \frac{1}{1 + \sum CCC_s}$. There-

fore, a class with no Coupling Component Costs will have a coupling factor of 1.0 (very desirable), while a class with many Coupling Component Costs will have a much lower coupling factor (less desirable). In the extreme the coupling factor will approach zero, since

$\lim_{n \rightarrow \infty} \frac{1}{n} = 0$, which is the least desirable case. If there are S classes in the system, the over-

all system coupling factor is given as $\frac{\sum_{c=1}^S CF_c}{S}$, or the arithmetic mean of the constituent coupling factors.

Some researchers have suggested that the coupling between two classes increases with the number of times one object references a method in the other class. In other words, *ceteris paribus*, if class *A* references a method in class *B* more frequently than a method in class *C*, the coupling from class *A* to class *B* is tighter than the coupling from class *A* to class *C*. It is further suggested that class *B* should be subjected to more stringent testing than class *C*. [6] This argument is rejected because a properly designed coupling between two object classes should carry no “volume” penalty. On the other hand, an improperly designed coupling does carry a penalty (cost) in the proposed metric, and it is imposed irrespective of the number of actual uses. Finally, the argument that class *B* should be more stringently tested is also rejected. While it is agreed that complex methods may be deserving of more stringent testing, there is no demonstrated correlation between frequency of use and complexity.

The proposed coupling metric does not consider the volume of references between objects, nor does it suggest that frequently-used methods/classes should be more stringently tested than other methods/classes. It does suggest, however, that methods/classes with relatively low coupling factors should be more closely examined to determine if the coupling is proper or if the coupling could be improved.

Metric Validation

The validation of the proposed metrics is modeled after that described by Basili, et al. [1], and employs a controlled study to determine if cohesion and coupling are an indicator of the fault-proneness of a system.

A group of forty Colorado Technical University graduate students is randomly divided into eight teams. The team members all have a comparable level of expertise and experience in the object-oriented programming language used, and each team is given the task of developing the same non-trivial application software. The application chosen is a Pilot Logbook that enables private and commercial pilots to record flight times in a variety of categories, and to retrieve and summarize the recorded flight times in a wide variety of ways. For consistency, each team has access to the same domain expert. The teams all use the Extreme Programming (XP) paradigm, and the domain expert is the “customer” for each

Programming (XP) paradigm, and the domain expert is the “customer” for each team. The team members are free to organize their teams as they wish (e.g. selecting “pair programmers”, etc.) and to reorganize themselves as they see fit. No attempt is made to impose a specific software design, or to dictate the number, or contents, of software releases, except that the “customer” will consistently relay the same requirements to each team.

At the completion of the development effort, the software of each team is electronically examined, and the metrics, as described above, are automatically tabulated. In addition, the finished programs will be provided to a group of forty “volunteer” pilots for a three-month use and testing period. Each volunteer pilot receives the program from only one team, but each team has the same number of users/testers.

One of the required features of the software is the ability to allow the user to easily report a suspected program error. At the completion of the three-month period, the problem reports are forwarded to the respective team, who determines, for each problem report, if it is an actual program error. Each authenticated error is ascribed to a primary class and method.

Since the response variable is binary, e.g. there was a fault or there wasn't, logistic regression is an appropriate tool to analyze the results.[1, 9] The details of logistic regression are beyond the scope of this paper, but it is a statistical technique used to show that a particular correlation is more than mere chance.

The data collected in this study comprises only the metrics automatically tabulated from the completed software, and the authenticated problem reports (with the associated responsible class and method). No data is collected with respect to how much time was spent on any given class or method, or on how much time was spent on the overall system. In addition, no data is collected on “bugs” found and corrected during testing, since it is believed that the number of such bugs is simply a function of the testing effort and is not necessarily indicative of a faulty class/method. This would, of course, be a good area for future research. Finally, the teams are not asked to actually correct the problems reported, nor are they asked to assign a “severity” to any problem report. For the purposes of this study, all problems carry the same weight, and are binary in nature (e.g. “yes” or “no”).

CONCLUSION

Cohesion and coupling are believed by many to be an important element in a “good” software design. While this makes sense on an intuitive level, there have been few empirical studies/experiments to support the intuition. In addition, even the definition of the terms is somewhat problematic, and relies upon vague terms such as “relatedness” and “strength of interconnections” to describe them. The measurement of cohesion is particularly difficult, and all metrics to date focus on the interaction and use of common instance variables, even though there is little evidence to suggest that this is a relevant component of cohesion. Until further experiments are performed in this area, it may very well be that we are “making important what we can measure.”

This paper describes two new metrics – one for cohesion and another for coupling – and describes a method whereby the metrics can be validated (or not). While the validation has not been performed, it is believed that it will demonstrate that cohesion is *not* an important component of a good object-oriented software design. It is further believed that the validation will demonstrate that coupling *is* an important component, and that the proposed metric is a good indicator of fault-prone methods/classes.

Bibliography

1. Basili, Victor R, Lionel Briand, and Walcélio L. Melo, *A Validation of Object-Oriented Design Metrics as Quality Indicators*, 1995.
<ftp.cs.umd.edu/pub/papers/papers/ncstrl.umcp/SC-TR-3443/CS-TR-3443.ps.Z>
2. Bieman, James M. and Kang Byung-Kyoo, *Cohesion and Reuse in an Object-Oriented System*, 1995. <ftp.cs.colostate.edu/pub/bieman/bieman-kang-ssr95.ps.gz>
3. Briand, Lionel and Jürgen Wüst. *The Impact of Design Properties on Development Cost in Object-Oriented Systems*. Proc. *Seventh International Software Metrics Symposium*. 2001: IEEE.
4. Chae, Heung Seok and Yong Rae Kwon. *A Cohesion Measure for Classes in Object-Oriented Systems*. Proc. *5th International Symposium on Software Metrics*. 1998. Bethesda, MD: IEEE.
5. Chidamber, Shyam R. and Chris F. Kemerer. *Towards A Metrics Suite For Object Oriented Design*. in *OOPSLA '91*. 1991: ACM.
6. Chidamber, Shyam R. and Chris F. Kemerer, *A Metrics Suite for Object Oriented Design*. *IEEE Transactions on Software Engineering*, 1994. 20(6): p. 476-493.
7. Clark, Bradford, Sunita Devnani-Chulani, and Barry Boehm, *Calibrating COCOMO II Post-Architecture Model*. IEEE, 1998. 6: p. 477-480.

8. Emam, Khaled El, et al., *A Validation of Object-Oriented Metrics*, 1999.
ai.iit.nrc.ca/pub/iit-papers/NRC-43607.pdf
9. Emam, Khaled El, *A Methodology for Validating Software Product Metrics*, 2000.
ai.iit.nrc.ca/pub/iit-papers/NRC-44142.pdf
10. Fenton, Norman E. and Shari Lawrence Pfleeger, *Software Metrics: A Rigorous & Practical Approach*. Second ed. 1997: PWS Publishing Company.
11. Hitz, Martin and Behzad Montazeri, *Chidamber and Kemerer's Metrics Suite: A Measurement Theory Perspective*. IEEE Transactions on Software Engineering, 1996. 22(4): p. 267-271.
12. Sommerville, Ian, *Software Engineering*. Third ed. 1989: Addison-Wesley Publishing Company.
13. Xia, Franck. *Module Coupling: A Design Metric*. Proc. 3rd Asia-Pacific Software Engineering Conference (APSEC '96). 1996: IEEE.