An Overview of Object-Oriented Design Metrics

R. Harrison, S. Counsell, R. Nithi
Department of Electronics and Computer Science,
University of Southampton, Southampton, SO17 1BJ, U.K.
email: rh@ecs.soton.ac.uk

Abstract

In this paper, we examine the current state in the field of object-oriented design metrics. We describe three sets of currently available metrics suites, namely, those of Chidamber and Kemerer, Lorenz and Kidd and Abreu. We consider the important features of each set, and assess the appropriateness and usefulness of each in evaluating the design of object-oriented systems. As a result, we identify problems common to all three sets of metrics, allowing us to suggest possible improvements in this area.

1. Introduction

Various sets of object-oriented metrics have been proposed as a means of assessing whether systems under investigation exhibit characteristics of quality software [6, 7, 14, 5, 15]. In this paper, three sets of such metrics are considered. The first set were proposed by Chidamber and Kemerer [7], the second by Lorenz and Kidd [15], and lastly, those proposed by Abreu [5]. All three sets attempt to capture the key elements of object-oriented software: *encapsulation* (information hiding), *abstraction* and *inheritance*.

In Sections 2, 3 and 4 respectively, each of these sets of metrics is analysed, and the important features of each highlighted. The intentions of the designers of the metrics, in terms of what the metrics were intended to measure, are also discussed. Various problems with the three sets considered are highlighted as a result, and these are then discussed (Section 5); possible future directions are then considered, taking account of these shortcomings (Section 6).

2. Description of Chidamber and Kemerer Metrics

In keeping with the key elements of object-oriented software, the set of six metrics developed by Chidamber and Kemerer (C&K) [7] attempt to identify certain design traits in object-oriented software, for example, inheritance, coupling and cohesion etc. The six metrics can be summarised as:

1. Weighted Methods per Class (WMC). This metric counts the number of methods in a class. WMC was designed to measure the complexity of a class. However, since C&K offer no definition of complexity, it is considered to be unity. WMC is therefore a measure of size, and equivalent to the number of methods in the class.

2. Depth of Inheritance Tree (DIT). This metric measures the maximum level of the inheritance hierarchy of a class; the root of the inheritance tree inherits from no class and is at level zero of the inheritance tree. DIT was intended to indicate the potential for reuse, and to indicate the complexity of the design.

3. Number of Children (NOC). This metric counts the number of subclasses belonging to a class. C&K suggest that the NOC can be used to indicate the level of reuse in a system, and hence be used as a possible indicator of the level of testing required.

4. Lack of Cohesion in Methods (LCOM). This metric purports to measure the lack of cohesion in the methods of a class. It is based on the principle that a variable occurring in many methods of a class causes that class to be less cohesive than one where the same variable is used in few methods of the class. As one would expect, C&K view a lack of cohesion as undesirable.

5. Coupling Between Objects (CBO). This metric measures the level of coupling between classes. Coupling between two classes is said to occur when one class uses functions or variables of another class. C&K suggest CBO as an indication of the effort needed for maintenance and testing. A high CBO is considered undesirable.

0-8186-7840-2/97 $10.00 © 1997 IEEE

6. Response For a Class (RFC). This metric counts the occurrences of calls to other classes from a particular class. In other words, the set of all methods which can be invoked in response to a message to an object of the class. C&K view RFC as an indication of class complexity (and hence a reflection of the testing effort required).

3. Description of Lorenz and Kidd Metrics

We now describe ten metrics proposed by Lorenz and Kidd (L&K). We note that many other metrics were suggested by L&K in [15]. However, the ten metrics described give a fair cross-section of the broad areas covered. Unlike the C&K metrics, most of the L&K metrics are direct metrics, and include more directly countable measures, e.g., the Number of Methods (NM) metric, and the Number of Variables (NV) metric. Although relatively simple to collect, doubt can be cast on the usefulness of such metrics because they give only a limited insight into the architecture of the system under investigation. For each of the metrics considered, L&K offered some justification for the existence of that metric and we include that justification in the following analysis. The ten metrics can be summarised as:

1. Number of Public Methods (PM). This simply counts the number of public methods in a class. According to L&K, this metric is useful as an aid to estimating the amount of work to develop a class or subsystem.

2. Number of Methods (NM). The total number of methods in a class counts all public, private and protected methods defined. L&K suggest this metric as a useful indication of the classes which may be trying to do too much work themselves; i.e., they provide too much functionality.

3. Number of Public Variables per class (NPV). This metric counts the number of public variables in a class. L&K consider the number of variables in a class to be one measure of its size. The fact that one class has more public variables than another might imply that the class has more relationships with other objects and, as such, is more likely to be a *key class*, i.e., a central point of co-ordination of objects within the system.

4. Number of Variables per class (NV). This metric counts the total number of variables in a class. The total number of variables metric includes public, private and protected variables. According to L&K, the ratio of private and protected variables to total number of variables indicates the effort required by that class in providing information to other classes. Private and protected variables are therefore viewed merely as data to service the methods in the class.

5. Number of Methods Inherited by a subclass (NMI). This metric measures the number of methods inherited by a subclass. No mention is made as to whether that inheritance is public or private. In a language such as C++, we have to consider the possibility that the inheritance may be private, Then, any classes using methods from a subclass would not necessarily have access to all of the inherited methods.

6. Number of Methods Overridden by a subclass (NMO). A subclass is allowed to re-define or override a method in its superclass(es) with the same name as a method in one of its superclasses. According to L&K, a large number of overridden methods indicates a design problem, indicating that those methods were overridden as a design afterthought. They suggest that a subclass should really be a specialisation of its superclasses, resulting in new unique names for methods.

7. Number of Methods Added by a subclass (NMA). According to L&K, the normal expectation for a subclass is that it will further specialise (or add) methods to the superclass object. A method is defined as an added method in a subclass if there is no method of the same name in any of its superclasses.

8. Average Method Size (AMS). The average method size is calculated as the number of non-comment, non-blank source lines (NCSL) in the class, divided by the number of its methods. AMS is clearly a size metric, and would be useful for spotting outliers, i.e., abnormally large methods.

9. Number of times a Class is Reused (NCR). The definition of NCR given by L&K is somewhat ambiguous. We assume the metric is intended to count the number of times a class is referenced (i.e., reused) by other classes. In this sense, we could view reuse in a similar way to coupling. We could then consider NCR as a measure of the extent of inter-class communication, and in this respect, a high value for NCR as undesirable.

10. Number of Friends of a class (NF). This metric measures, for each class, the number of friends of that class. Friends allow encapsulation to be violated, and as such should be used with care. A high number of friends within a class could indicate a potential design flaw, an oversight in design, which has filtered through to the coding stage; we note in passing that friends are a concept specific to the C++ language. NF is a measure of class coupling, since friends may rely on a particular class (or classes) to operate properly.

4. Description of Abreu Metrics

The set of six metrics developed by Abreu [5] were intended to be *design* metrics. The emphasis behind the development of the metrics is on the features of inheritance, encapsulation and coupling. The six metrics can be summarised as:

1. Polymorphism Factor (PF). This metric is based on the number of overriding methods in a class as a ratio of the total possible number of overridden methods. Polymorphism arises from inheritance, and Abreu claims that in some cases, overriding methods reduce complexity, so increasing understandability and maintainability.

2. Coupling Factor (CF). This metric counts the number of inter-class communications. There is a similarity here with the NCR metric of L&K. Abreu views coupling as increasing complexity, reducing both encapsulation and potential reuse and limiting understandability and maintainability.

3. Method Hiding Factor (MHF). This metric is the ratio of hidden (private or protected) methods to total methods. As such, MHF is proposed as a measure of encapsulation.

4. Attribute Hiding Factor (AHF). This metric is the ratio of hidden (private or protected) attributes to total attributes. AHF is also proposed as a measure of encapsulation.

5. Method Inheritance Factor (MIF). This metric is a count of the number of inherited methods as a ratio of total methods. There is a similarity here with the NCR metric of L&K. Abreu proposes MIF as a measure of inheritance, and consequently as a means of expressing the level of reuse in a system. It could also claim to be an aid to assessment of testing needed.

6. Attribute Inheritance Factor (AIF). This metric counts the number of inherited attributes as a ratio of total attributes. Just as for the MIF, Abreu proposes AIF as a means of expressing the level of reuse in a system. It is claimed, however, that too much reuse causes a deterioration in understandability and testability.

5. Analysis of the three metric suites

Each of the individual metrics in the three metric suites just described purport to measure some attribute(s) of an object-oriented system. Various shortcomings emerge when we begin to consider criteria important in designing, using and interpreting object-oriented metrics for real systems. The first consideration concerns that of *validity*.

5.1. Validity of Metrics

The theoretical approach to the validation of metrics requires us to clarify what attributes of software we are measuring, and how we go about measuring those attributes [12, 4, 1, 8]. A metric must measure what it purports to measure.

Fenton [9] describes the *representation condition*, satisfaction of which is the pre-requisite for any metric to be viewed as valid. The representation condition states that any measurement mapping must map entities into numbers, and empirical relations into numerical relations, such that those relations are preserved. In other words, our observations in the real world must be reflected in the numerical values we obtain from the mathematical world.

Kitchenham et al. describe a list of features of metrics which must hold for that metric to be valid [12]. A direct metric must not exhibit any unexpected discontinuities, should use the appropriate measurement scale, should be dimensionally consistent and be based on an explicitly defined model of the relationship between attributes.

5.2. Example 1

Consider the Weighted Methods Per Class metric (WMC) of C&K. It was intended to be a measure of complexity, but, in the absence of any definition of complexity, is a measure of class size. We can not view WMC as an indicator of the effort to develop a class, since a class containing a single large method may take as long to develop as a class containing a large number of small methods. The same can be said of the L&K Public Methods (PM) metric.

5.3. Example 2

Consider the Depth of Inheritance Tree metric (DIT) of C&K. It was intended to be an indication of the potential for reuse, but is actually a direct count of the levels in an inheritance hierarchy. It is easy to envisage an inheritance hierarchy which was very deep and yet reused the same number of methods as a shallow, wide hierarchy; using the DIT metric in this case would, however, give very different answers for the two systems.

5.4. Example 3

Consider three metrics: the Response for a Class (RFC) metric of C&K, the Number of Methods Inherited by a subclass (NMI) metric and the Number of times a Class is Reused (NCR) metric, the latter two both of L&K. In each, there is some ambiguity in exactly what the respective designers meant the metric to measure. This forces the user to guess what they think the metric was intended to measure.

Clearly, there is a need for a formal definition of each metric so that there is no ambiguity in its interpretation. It is difficult to validate a metric if its definition is ambiguous.

5.5. Example 4

Consider the Number of Variables per class (NV) metric of L&K. It is claimed that the ratio of private and protected variables to total number of variables is an indication of the effort required by that class to provide information to other classes. However, some would claim that all variables should be be hidden, so maximising encapsulation.

5.6. Example 5

Consider the Polymorphism Factor (PF) metric of Abreu. It is defined as the ratio of overridden methods to total possible overridden methods. Hence, if there is no inheritance in the system under consideration, then the denominator of the metric calculation is zero (giving an infinite value for the metric itself). This would seem to contradict a feature stated by Kitchenham et al [12], i.e., that a metric should not have any unexpected discontinuities.

5.7. The need for empirical evaluation

Numerous authors have suggested that theoretical validation of any set of metrics should not be the only support for a proposed set of metrics. It should be accompanied by empirical evaluation, using proven statistical and experimental techniques; in this way, the practical applicability of any new metrics in the field can be assessed [4, 1, 2, 17]. For example, in [4], Briand et al. describe metrics for cohesion and coupling, showing how an in-depth empirical evaluation demonstrated and supported the usefulness and significance of the set of metrics they proposed. Again, we give a number of examples to illustrate the ideas.

5.8. Example 1

Consider the Number of Friends (NF) metric of L&K which suggests that too many friends indicates a design flaw, and is indicative of oversights in design. This thesis could be supported by perhaps trying to identify a relationship between NF and the number of modification requests (MR's). A high number of MR's indicates a poor design and so the existence of a relationship between the two would add weight to this claim.

5.9. Example 2

Consider the Coupling Factor (CF) metric of Abreu. Coupling is viewed as undesirable, and is claimed to increase complexity and reduce both encapsulation and potential reuse. The thesis regarding increased complexity would be supported more strongly if an empirical evaluation were performed to identify any correlation between CF and, perhaps, a subjective measure of the complexity of each class (provided by the system designer). The thesis regarding encapsulation would be supported by identifying a relationship between CF and private/protected methods of a class; the reuse thesis would be supported by identifying a relationship between CF and a measure of reuse, such as the number of methods reused.

5.10. Example 3

Consider the Number of Children (NOC) metric of C&K. It is claimed to give an indication of the level of testing required. One way to support this thesis might be to investigate the relationship between NOC values and the testing times for each class. Such times can be easily and accurately collected during testing.

5.11. Data Collection

One of the main problems with the three sets of metrics proposed is the difficulty in collecting raw data from the code in order to calculate the metric values. For large systems, collection of the more involved metrics becomes prohibitively time-consuming. For example, calculation of the Lack of Cohesion in Methods (LCOM) metric (C&K), requires careful consideration of the use of variables in a class, and so is only practical for systems with a small number of classes; similarly for the Coupling Between Objects (CBO) metric (C&K). We also note that, in systems with no inheritance, the collection of metrics such as C&K's, L&K's and Abreu's becomes relatively trivial. Although this makes the metrics collection easier, our understanding of the system being analysed is then limited by a large number of metrics with values of zero.

5.12. Tool Support

There is very little tool support for metrics collection. The tool used to produce some of the values in [10], and which collects Abreu's set of metrics, tends to consume a lot of machine resources, and it is not entirely clear how the results obtained have been calculated. Performing a manual collection (where possible) from the code to validate the results is a painstaking, yet necessary task.

5.13. The over-emphasis on code metrics

Many of the metrics outlined are simply **code** metrics in the sense that they are measures of the code's characteristics. The WMC and LCOM metrics of C&K are two prime

examples of this. Yet, the three set of metrics studied claim to be high-level design metrics, and to indicate features of object-oriented systems design. It would be more useful to have metrics which measured the quality of the design at a much higher level of abstraction. Flaws in the design could then be identified before they filtered through to the code. This might just possibly alleviate the maintenance and testing problems with which all systems seem to suffer. The many object-oriented design and modelling techniques currently available could be used as a basis for these high-level metrics [16, 3, 19, 18].

5.14. Quality models

The metrics described give suggestions as to what aspects of quality object-oriented software they would be useful for measuring. Quality factors such as reusability, maintainability and testability are frequently quoted.

However, no concrete notion of what constitutes quality is provided by the designers of the metrics studied. Since quality for the systems developer is perceived differently to quality for the manager or end user, a set of metrics must make clear what it is they are trying to measure and who they are directed at. Metrics based on a properly defined quality model, incorporating ideas from many of the current standard quality models [11, 13] would clarify what quality aspects were being considered, and why they were being considered.

6. Future Directions

From our analysis of the three metric suites considered, a number of problems have been shown to exist with currently available metrics:

1. many of the metrics are suspect in terms of their validity, and often do not measure the attributes of software they purport to measure.
2. there is ambiguity in some of the definitions.
3. very little empirical evaluation exists to support the claims made of the individual metrics.
4. in the absence of tool support, manual collection of metrics can be cumbersome and time-consuming.
5. there is currently an over-emphasis on code metrics.
6. a proper definition of quality, based on rigorously defined quality models has been largely ignored.

Future directions in this field are clearly for metrics which are valid (and unambiguous), are aimed at a higher level of abstraction (the design stage of development), are based on a rigorous model of what constitutes software quality and, finally, are supported by appropriate tools and proper empirical evaluation.

7. Conclusion

In this paper, we have described three sets of object-oriented metrics, the problems associated with those metrics, and suggestions for future directions in this field. The main point to note is that, just as our use and understanding of object-oriented systems is still in its formative stages, so the same is true in the field of object-oriented metrics. As a way forward, we should therefore seek to learn as much as possible from the problems we are currently encountering in this area, and adopt solutions to overcome these problems.

8. Acknowledgements

This work is supported by UK EPSRC project GR/K83021.

References

[1] V. R. Basili, L. Briand, and W. Melo. A validation of OO design metrics as quality indicators. *Technical Report CS-TR-3443*, 1995.

[2] V. R. Basili and H. D. Rombach. The TAME project: Towards improvement-oriented software environments. *IEEE Transactions on Software Engineering*, 14(6):758–773, 1988.

[3] G. Booch. *Object-oriented design with applications*. Benjamin-Cummings, 1991.

[4] L. Briand, S. Morasca, and V. R. Basili. Defining and validating high-level design metrics. *Technical Report CS-TR-3301*, 1994.

[5] F. Brito e Abreu, M. Goulao, and R. Esteves. Toward the design quality evaluation of OO software systems. In *5th Int Conf on Software Quality*, 1995.

[6] S. R. Chidamber and C. F. Kemerer. Moose: Metrics for object oriented software engineering. In *Workshop on Processes and Metrics for Object Oriented Software Development, OOPSLA '93, Washington*, 1993.

[7] S. R. Chidamber and C. F. Kemerer. A metric suite for object oriented design. *IEEE Transactions on Software Engineering*, pages 467–493, 1994.

[8] N. E. Fenton. Software measurement: a necessary scientific basis. *IEEE Transactions on Software Engineering*, 20(3):199–206, 1994.

[9] N. E. Fenton and P. S. L. *Software Metrics, A Rigorous and Practical Approach*. International Thomson Computer Press, 1996.

[10] R. Harrison, S. Counsell, and R. Nithi. Empirical assessment of object-oriented design metrics. In *Proceedings of Empirical Assessment in Software Engineering (EASE) '97, Keele, UK*, 1997.

[11] ISO/IEC. Joint technical committee: Information technology - software product evaluation - quality characteristics and guidelines for their use. International standard, ISO/IEC, 1991.

[12] B. A. Kitchenham, S. L. Pfleeger, and N. Fenton. Towards a framework for software measurement validation. *IEEE Transactions on Software Engineering*, 21(12):929–944, 1995.

[13] B. A. Kitchenham, J. D. Walker, and I. Domville. Test specification and quality management - design of a qms subsystemfor quality requiremnets specification. Project Deliverable A27, Alvey Project SE/031, Nov 1986.

[14] W. Li and S. Henry. Maintenance metrics for the object-oriented paradigm. In *Proceedings of the First International Software Metrics Symposium, Baltimore Maryland*, pages 52–60, May 1993.

[15] M. Lorenz and J. Kidd. *Object-oriented Software Metrics*. Prentice Hall Object-Oriented Series, 1994.

[16] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorenson. *Object-oriented modelling and design*. PHI, 1991.

[17] N. F. Schneidewind. Methodology for validating software metrics. *IEEE Transactions on Software Engineering*, 18(5):410–422, 1992.

[18] S. Shlaer and S. Mellor. *Object-Oriented Systems Analysis: Modelling the World in Data*. Prentice Hall, 1988.

[19] R. Wirfs-Brock, B. Wilkerson, and L. Wiener. *Designing Object-Oriented Software*. Prentice Hall, 1990.