ISSN No. 0976-5697

Volume 2, No. 1, Jan-Feb 2011

International Journal of Advanced Research in Computer Science

RESEARCH PAPER

Available Online at www.ijarcs.info

Analyzing Software Evolution Using the MOOD Metric Set

Kuljit Kaur* and Hardeep Singh
Deptt. Of Computer Science and Engineering,
Guru Nanak Dev University, Amritsar
India
kuljitchahal@yahoo.com

Abstract: A reusable component once developed can be used several times in different applications. It is tested before every use and this repeated testing removes defects, and increases (re)user's confidence in the quality of the software component. For a reusable component, quality of the component is expected to improve as it matures. However, it is also believed that the quality of a program degrades if it is not managed in successive version releases. In this research, an attempt has been made to identify metrics trends in successive evolutions of a reusable component in order to understand the trends in its design quality. Metrics defined in the MOOD metric set have been used to measure system level properties of the software component.

Keywords: Component Based Software Engineering, Software Components, Software Metrics, MOOD Metrics, Software Evolution

I. INTRODUCTION

In component based software development, selection of suitable components at the proper time is a prerequisite to achieve objectives of improved product quality within time and budget constraints. Component evaluation is a critical activity in the component selection process. A component has to be evaluated technically (functionality and quality) as well as non-technically (cost, and vendor support etc.) [1]. Several component quality attributes such as reusability, and maintainability depend upon the structural properties of its design [2]. One method of component evaluation is to evaluate its design for various concepts such as complexity, coupling, and cohesion using software metrics.

Software component level metrics fall into two categories: component interface related metrics, and component structure related metrics. Metrics in the former category are collected from information available in component interfaces and are applicable to black box components [3]). In the latter category, metrics are collected from internal structure (design or code) of a component [4-6] and are applicable to white box components only.

Researchers define metrics for internal structure in light of the guidelines prescribed for designing reusable components. Poulin *et al.* discusses several internal properties of the structure of a program that make it reusable [7]. Modern technologies such as object oriented software development concentrate on building reusable artifacts from the early stages of life cycle rather than at later stages [8]). Object oriented programming provides features to build generic software components which can be reused in multiple applications with little or no modification [9]. The application of object-oriented design-principles like modularity, abstraction, and de-coupling ("good design") lead to better maintainability and reusability. However, the object-oriented paradigm alone does not guarantee good design. Developers have to understand design principles and to check whether those have been obeyed.

This paper analyzes a reusable software component with the help of the system level metrics, popularly known as the MOOD metric set, proposed by Abreu *et al.* [10] Next Section discusses the object oriented concepts and the related metrics from the MOOD metric set. Third section of the paper gives details about the data collection. Fourth section provides the metrics analysis. Fifth section concludes the paper.

II. OBJECT ORIENTED CONCEPTS AND METRICS

This section gives an account of the metrics to measure the structural properties of an object oriented software component. Metrics are categorized according to the elements of the object oriented paradigm they measure.

A. Inheritance

Inheritance is the relationship between two or more classes in which definition of a new class is based on the definition of existing classes. The existing class is called the super/parent/base class and the newly defined class is called the subclass/child/derived class. A subclass inherits features from its super class and adds new features to refine the definition of the super class. Inheritance represents 'is a kind of' relationship so an instance of a sub class is also an instance of its super class. A subclass can also respond to all the operations to which its super class can respond. Method Inheritance factor (*MIF*) and Attribute Inheritance Factor (*AIF*) are the system level metrics that measure the usage of inheritance mechanism [10].

B. Polymorphism

Polymorphism is the ability of a message to take many forms depending upon the message sender. The operation to be called to answer the message may be decided at compile-time or at run-time. So polymorphism can be static (compile time) or dynamic (run time). Also polymorphism can be adhoc or universal. If different classes (may not be from the same hierarchy) use just the same name for a similar kind of operation it is called adhoc polymorphism. However, if classes in the same hierarchy implement the same operation in different ways, it is known as universal polymorphism. Implementation of an operation may vary along the hierarchy of classes. At the system level, the Polymorphism Factor

(*PF*) metric from the MOOD metric set [10] measures the polymorphic behaviour of classes taken together.

C. Information Hiding

In object oriented paradigm, encapsulation means grouping data structures with the methods that manipulate them. It is advised that parts of a complex system should not depend on the internal details of one another [11]. Clients should access the services of the class through message passing. There is no need to know internal details/ implementation of the class elements. Parnas introduced the idea of information hiding [12]. Encapsulation and information hiding both lead to a stable software design as changes are localized and changes to a class (if it does not affect its behaviour) do not affect clients of the class. Software becomes flexible and can accommodate changes easily. Method Hiding Factor *(MHF)* and Attribute Hiding Factor *(AHF)* measure the degree of information hiding in a system [10].

D. Coupling

In any system, components of the system cannot exist in isolation. They have to depend upon one another in order to support the behaviour of the system. Coupling refers to the inter-dependencies in components of the system. Classes depend upon one another to provide functionality of the system. Two classes are said to be tightly (loosely) coupled if they depend highly (lowly) upon details of each other. A Loosely coupled class is easy to understand, and change because understanding such a class does not require understanding many other related classes and changing such a class does not have any impact on many other classes. It is also easy to reuse such a class. So loose coupling between classes improves maintainability and reusability of the system. In an object oriented design, coupling metrics measure the interdependencies of different classes. A design with a large number of inter class dependencies (coupling) is weak and fragile. *CF* metric measures coupling between classes at system level [10].

III. DATA COLLECTION

The metric data is collected from an open source reusable component, JFreeChart, available in the repository of open source software at www.sourceforge.net/jfreechart. It is a JAVA based software for creating different types of charts/graph. The software component JFreeChart is downloaded from its home page. All the versions of the component starting from the first one released in November, 2000 are available at this link. In this research only 43 versions starting from JFreeChart 0.5.6 to JFreeChart 1.0.11 are considered.

Borland Together is a product of Borland Software Corporation. Borland Together can be used for modeling new applications as well as for extracting design information from the existing ones. It also facilitates quality assurance by providing metrics and audits at model and code level to analyze and assess an application's quality. In addition to this, Together provides many more features (for details see http://techpubs.borland.com/together/2008R2/EN/Together.pdf).

Together includes a variety of metrics to analyze an application at different levels: system, package, and class level. Here only system level metrics are collected using the Borland Together tool.

IV. METRICS ANALYSIS

System level metrics are the metrics which measure the properties of a system at the highest level of abstraction. In this category, this study includes metrics from the MOOD metric set [10]. This set has metrics to measure the basic properties of an object oriented design such as encapsulation, inheritance, polymorphism, and coupling. It is believed that these mechanisms, if incorporated in the design of a software product, help to make it easy to reuse and maintain [13]. But use of these features in a design depends upon the abilities of its designer. It is important to correlate improvements in software quality with the use of these mechanisms.

System level metrics for different versions of the software component were collected. Trends in the metric values are discussed next:

A. Method Hiding Factor (MHF) and Attribute Hiding Factor (AHF)

MHF and *AHF* represent average amount of class members (attributes or methods) hidden from other classes in a system. If all members of all the classes are hidden, then *MHF* and *AHF* both are 100% for the system. But this could not be possible practically. A class cannot exist in isolation in a system. It has to communicate with other classes to support the functionality of the system. It has to declare some of its methods as public. Therefore *AHF* may attain value 100% (and it is ideal too), but *MHF* should not. Number of visible methods of a class indicates its functionality. Larger is the value, more will be the functionality. High values of *MHF* indicate very less functionality. On the other hand, if all members of all the classes are public, then *AHF* and *MHF* both are 0% for the system. This is also an alarming situation. A large number of public members of classes increase the probability of errors in a system.

An acceptable range of 8% to 25% is suggested for *MHF*[1]. In another study of MOOD metrics on 9 commercial projects, *MHF* takes values in this range [14].

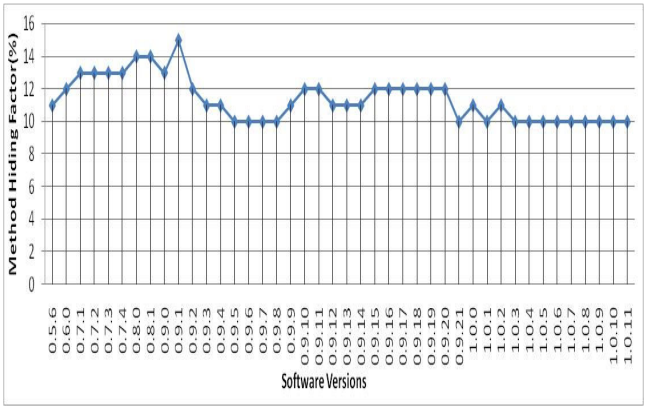

Figure.1: Method Hiding Factor (*MHF*) Metric Trend

It could be observed from Figure 4.4, that the method hiding factor (*MHF*) metric remains within the prescribed limits for all the releases of the software component. *MHF* values in the lower range may be due to the fact that a proper top down decomposition process has not been followed for implementing abstractions in the system. On the other hand in Figure 2, attribute hiding factor (*AHF*) was initially low but it has improved over time. *AHF* is close to the optimal value. So *MHF* and *AHF* both show positive trends for this

software component. It can be said that the design of the software component adheres to the concept of information hiding.

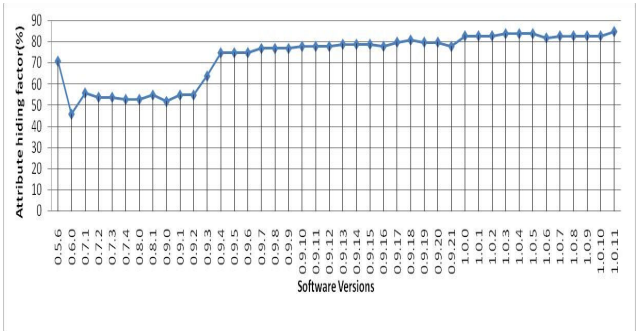

Figure.2: Attribute Hiding Factor (*AHF*) Metric Trend

B. Method Inheritance Factor (MIF) and Attribute Inheritance factor (AIF)

MIF and *AIF* measure the extent to which individual classes of a system inherit properties from their respective base classes. *MIF* (*AIF*) is the ratio of the sum of inherited methods (attributes) in all classes of a system to the total number of available methods (attributes) in all the classes. Systems in which classes inherit a large number of properties have large values of *MIF/AIF*.

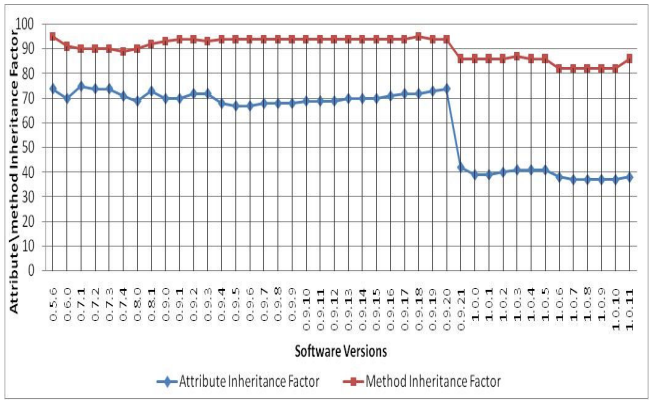

Figure.3: Attribute Inheritance Factor and Method Inheritance Factor Metrics Trends

All the releases show sufficient amount of inheritance. In Figure 3, *MIF* takes values in the range 80% to 95%, and *AIF* varies from 37% to 75%. These high values indicate satisfactory use of method inheritance. However in recent versions, there is a significant reduction in values of *AIF* with a very sharp decline from version JFreeChart 0.9.20 to JFreeChart 0.9.21. It may be due to increase in average class size as well as the number of classes of the software component over the period of time. As the denominator in case of *AIF* (*MIF*) metric is the sum of attributes (methods) of all classes in a system, increase in the value of the denominator may have resulted in decreasing trend for the metric values.

C. Polymorphism Factor (PF)

Polymorphism means having the ability to take several forms. For object-oriented systems, polymorphism allows the implementation of a given operation to be dependent on the object that contains the operation. An operation can be implemented in different ways in different classes. Classes with polymorphic operations are easier to extend and modify. The polymorphism factor (*PF*) metric is defined as the ratio of the actual number of different polymorphic situations to the maximum number of possible distinct polymorphic situations for all classes in a system.

In successive versions of this software component, *PF* takes values from 4% to 10%. Decreasing values of *PF* show less use of dynamic binding. Figure 3 shows that *MIF* is very high, i.e. there is considerable use of method inheritance. But decreasing values of *PF* in Figure 4 indicate that inherited methods are not extensively redefined in the subclasses. It is not desirable to redefine a large number of inherited methods as it indicates that hierarchy is created out of convenience rather than a natural one. Moreover the exact behaviour of a program in this regard can be studied with the help of dynamic metrics [15].

D. Coupling Factor (CF)

Two or more classes are said to be coupled if they exchange messages or have other kind of relationships such as inheritance, aggregation, or association. More couplings in classes means that classes are more inter-dependent, difficult to understand and therefore harder to change and repair. Coupling factor is the ratio of actual number of couplings to the maximum possible pair wise couplings in a system. It does not include inheritance based class relationships.

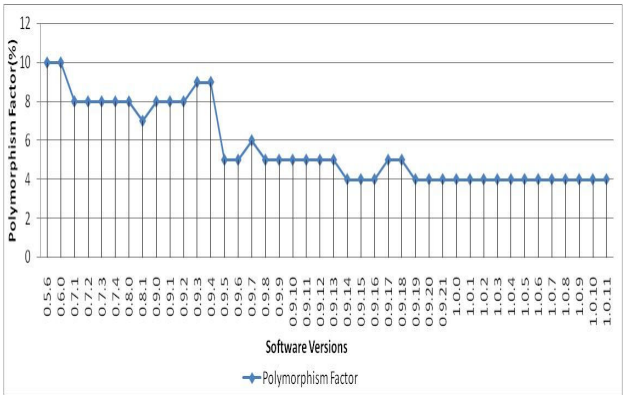

Figure.4: Polymorphism Factor (*PF*) Metric Trend

Metric value for the successive releases has decreased gradually except one peak that too in the second release only (Figure 5). *CF* took value 2 in JFreeChart version 0.9.4 and remained at that level for long time. After version JFreeChart 0.9.16, value of the metric *CF* is further reduced to 1 and has remained there till JFreeChart 1.0.11. Low values of *CF* indicate that classes communicate less with other classes which are not in the same

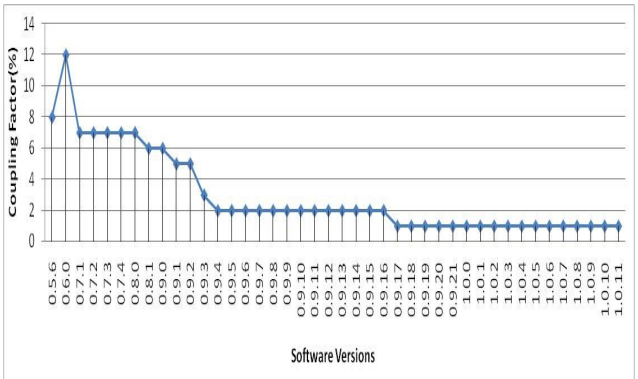

Figure.5: Coupling Factor (*CF*) Metric Trend

Inheritance hierarchy (as *CF* does not include inheritance based coupling). Excessive coupling between classes across the inheritance hierarchies does not indicate a good design as it is difficult to understand and modify for future extensions.

V. CONCLUSIONS

Metrics defined in the MOOD metric set have been used to measure system level properties of the software component. *MHF* and *AHF* measure the extent to which classes hide their information from other classes in the system. *CF* measures the degree of interclass coupling in the system. *MIF* and *AIF* measure the usage of inheritance mechanisms. *PF* measures the ability of the classes to behave in polymorphic ways. System level metrics attribute hiding factor (AHF) and method hiding factor (MHF) show that the software component incorporates the concept of information hiding. As the component evolves, AHF almost achieves its ideal value i.e. 100%. MHF is high for the initial releases and as the functionality increases MHF decreases in successive versions. Attribute inheritance factor (AIF) and method inheritance factor (MIF) also show significant use of inheritance. Polymorphism factor (PF) also follows a downward trend across the releases which shows a controlled overriding of the inherited features. The metric coupling factor (CF), for measuring coupling between classes, also shows a decreasing trend.

VI. REFERENCES

[1] Brereton, P. and Budgen, D. (2000). Component-Based Systems: A Classification of Issues, IEEE Computer 33(11): 54-62.

[2] Cai et al., 2000 Cai, X., Lyu, M. R., Wong, K. and Ko, R. (2000). Component-Based Software Engineering: Technologies, Development Frameworks, and Quality Assurance Schemes. Proceedings of Seventh Asia-Pacific Software Engineering Conference (APSEC'00). pp 372. Singapore.

[3] Washizaki, H., Hirokazu, Y. and Yoshiaki, F. (2003). A Metrics Suite for Measuring Reusability of Software Components. Proceedings of the 9th International Symposium on Software Metrics. pp: 211-223. Sydney, Australia.

[4] Gui and Scott, 2009 Gui, G. and Scott, P. (2009). Measuring Software Component Reusability by Coupling and Cohesion Metrics. Journal of Computers 4(9): 797-805. Academy Publishers.

[5] Choi, M., Lee, J. and Ha, J. (2006). A Component Cohesion Metric Applying the Properties of Linear Increment by Dynamic Dependency Relationships Between Classes. In Gavrilova, M. et al. (Eds.): International Conference on Computational Science and Applications (ICCSA 2006). Lecture Notes in Computer Science 3981. pp 49 – 58. Springer-Verlag Berlin Heidelberg.

[6] Wu et al., 2007 Wu, F. and Yi, T. (2007). A Structural Complexity Metric for Software Components. Proceedings of the 1st International Symposium on Data, Privacy, and E-Commerce (ISDPE 2007). pp161-163. Chengdu, China. IEEE Computer Society Press.

[7] Poulin, 1994 Poulin, J. (1994). Measuring Software Reusability. Proceedings of 3rd International Conference on Software Reuse. pp. 126-138. Rio de Janeiro, Brazil.

[8] Fauzi et al., 2004 Fauzi, A. and Du, W. (2004). Towards Reuse of Object-Oriented Software Design Models, Information and Software Technology 46:499–517.

[9] Sametinger, 1997 Sametinger, J. (1997). Software Engineering with Reusable Components, Springer, -Verlag New York, Inc., USA.

[10] Abreu et al. Abreu, F.B. and Melo, W.(1996). Evaluating the Impact of Object Oriented Design on Software Quality. Proceedings of the 3rd International Symposium on Software Metrics (Metrics'96), pp 90-99, Berlin, Germany.

[11] Ingalls, 1981 Ingalls, D. (1981). Design Principles behind Smalltalk, BYTE Magazine, August 1981.

[12] Parnas (1972 Parnas, D. (1972). On the Criteria to be Used in Decomposing Systems into Modules. Communications of the ACM 15(12): 1053-1058.

[13] Rumbaugh et al., 2002 Rumbaugh, J., Blaha, M, Premerlani, W., Eddy, F. and Lorensen, W. (2002). Object Oriented Modeling and Design. Pearson Education, Prentice Hall, India.

[14] Harrison et al., 1998a Harrison, R., Counsell, S.J. and Reuben,V.N. (1998). An evaluation of the MOOD set of object-oriented software metrics, IEEE Transactions on Software Engineering 24(6): 491–496.

[15] Yacoub, S.M., Ammar, H.H. and Robinson, T. (1999). Dynamic metrics for object oriented designs. Proceedings of Sixth International Symposium on Software Metrics, USA. IEEE Computer society. pp 50-61. Boca Raton, USA.