International Journal of Technical Research and Applications e-ISSN: 2320-8163,
www.ijtra.com Volume 6, Issue 2 (MARCH-APRIL 2018), PP. 92-97

STUDY OF VARIOUS QUALITY METRICS SUITABLE FOR THE OBJECT ORIENTED ENVIRONMENT

Sharqua Reyaz[1], Dipti Ranjan[2],
Department Of Computer Science
[1]Lucknow Institute of Technology, Lucknow
[2]Lucknow Institute of Technology, Lucknow
Uttar Pradesh, India

***Abstract*— Software metrics is broad terms for all those actions which entails some degree of software measurement and are anticipated to measure the software quality as well as performance characteristics quantitatively. These can serve as measures of software products for the purpose of comparison, fault prediction, cost estimation and forecasting. This study is based on the data from a large open source project The JFreeChart available at one of the largest storehouses of open source projects www.sourceforge.net. In this paper we study over 57 versions of this project released in the time period from 2000 to 2016.**

***Index Terms*— OO Metrics, S/w Metrics, S/w Quality.**

I. INTRODUCTION

IEEE defines a quality factor as "a management-oriented attribute of a software that contributes to its quality". A metric is a measurement function, and a software quality metric is a "function whose inputs are software data and whose output is a single numeric value that can be interpreted as a degree to which software possesses a given attribute that affects its quality".

The true value of product metrics comes from their association with measures of important external quality attributes. An external attribute is measured with respect to how the product relates to its environment. Examples of external attributes are testability, reliability, maintainability etc.

Software quality assurance is one of the most important components in software project management. Research on various perspectives of software quality and related activities has been conducted for several decades, and many conclusions and practices have been presented to improve software quality. One aspect of the research in this area is to establish software quality estimation models that could be used at the early stages of a project to estimate the quality level. The estimation results can act as a guideline to enhance the quality assurance performance.

A. Software Metrics

Software metrics are quantifiable measures that could be used to measure different characteristics of a software system or the software development process.

Measurement in the physical world can be categorized in two ways – direct measures and indirect measures. Software metrics can be categorized similarly. Direct measures of the software engineering process include cost and effort applied. Direct measure of the product includes line of the product, execution speed, memory size and defect reporting over some set period of time. Indirect measures of the product include functionality, quality, complexity, efficiency and reliability etc. software engineering is a stable, quantitative engineering discipline. Its stability arises from the wide range of metrics evolved by software engineers to measure various aspects of the software. The advantage of metrics is that you can measure in quantitative terms the different aspects of software that need evaluation on an ongoing basis for estimation.

B. Measuring Quality

Measurement enables to improve the software process, assist in the planning, tracking the control of a design. A good software engineer uses measurements to assess the quality of the analysis and design model, the source code, the test cases, etc. What does quality mean?

Quality refers to the inherent or distinctive characteristics or property of object, process or other thing. Such characteristics or properties may set things apart from other things, or may denote some degree of achievement or excellence. Many quality measures can be collected from literature, the main goal of metrics is to measure errors and defects. The following quality factor should have every metric.

The main quality characteristics (Reusability, Reliability, Complexity and Maintainability) are available in almost all quality models. However, researchers differ while choosing sub characteristics under these characteristics.

II. LITERATURE REVIEW

Ferreira et al. presented a study carried out on a large sample of object-oriented, open-source programs. They analyzed data from 40 programs developed in Java, including tools, libraries and frameworks, of varying sizes and from 11 application domains, in a total of more than 26,000 classes. From the achieved results, they suggested thresholds for six object-oriented software metrics: COF, LCOM, DIT, afferent couplings, number of public methods and number of public fields. The study concluded that values of those metrics, except DIT, can be modeled by a heavy-tailed distribution. This property means that, for most metrics, there is a low number of occurrences of high values and a far higher number of occurrences of low values. Values of DIT can be modeled by the Poisson distribution, having mean value 2. Based on the most commonly values found in practice, They derived general thresholds for object-oriented software metrics, and thresholds by application domain, size and type (tool, library and framework) of software system. As they did not discover pertinent difference among them, we suppose that the general thresholds can be applied to OO software in general. The recognized thresholds were evaluated by means of two experiments. The outcomes of this evaluation point to that the proposed thresholds can assist to recognize classes which defy design principles. Furthermore, the attained results suggest that the thresholds can efficiently assess a design as good when it actually is. The proposed thresholds were derived from common practice.

Goel and Bhatia analyzed, object-oriented metrics have been measured for three C++ programs under the categories of inheritance, coupling and cohesion. The metrics have been analyzed and used to understand the various characteristics of the object-oriented systems. The first conclusion that can be drawn from this study is that all the programs show good use of object-oriented features and result in reusable classes. It has also been found that out of the three features, Multilevel Inheritance has more impact on reusability. This study hence not only helps to get some understanding of the object-oriented systems but also proves that the metrics are good at evaluating the object-oriented system.

Goel and Bhatia analyzed, the faults can differ significantly in their impact on the operation of a software system. It would be valuable to use OO design metrics to help to identify the fault-proneness of classes when the impact of faults is taken into account. Based on a public domain data set ivy1.1, log4j1.0 and ant1.3 provided by NASA, we employed the statistical logistic regression method to investigate the fault-proneness prediction usefulness of OO design metrics with regard to high and low impact faults. They analyzed six OO design metrics from Chidamber and Kemerer's (1994) metrics suite and one size metric LOC from Halstead Metrics. Our main results are summarized as follows: The CBO, WMC, RFC, LCOM and LOC metrics are statistically significant across fault impact, while DIT and NOC are not significant for any fault impact. The fault-proneness prediction capabilities of these metrics differ greatly depending on the fault impact used. When applied to the classification of classes as fault-prone and not fault-prone in terms of high/low impact faults, the logistic regression models based on these metrics achieve a performance comparable to previous studies.

A clear understanding of the definitions of these complexity metrics and a promise of their relevance in improving the outcomes of software development projects let to a body or research primarily focusing on the validation of these metrics.

III. METHODOLOGY

The software JFreeChart is downloaded from its home page www.sourceforge.net/jfreechart. In this research several versions starting from JFreeChart are considered. The procedure followed for data collection is as follows:

1) Download the source code of the software component and reverse engineer it to get the design information. BOUML (www.bouml.free.fr) is the tool used for the reverse engineering process. BOUML generates the output in the form of an XMI (XML Metadata Interchange) file.

2) Input the XMI file to the SDMetrics tool (www.sdmetrics.com) to collect the metrics. SDMetrics tool collects the metrics data and produces it in the CSV format. It collects metrics at class as well as package level.

IV. ANALYSIS OF OO METRICS

This research analyzes metrics at three levels- System, Package and Class level.

A. System Level Metrics

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 (Abreu et al., 1996). 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. 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 JfreeChart software were collected.

Trends in the metric values are discussed next:

1) 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. In another study of MOOD metrics on 9 commercial projects, MHF takes values in this range (Harrison et al., 1998a).

Fig.1: Method Hiding Factor (MHF) Metric Trend

It could be observed from Figure 1, 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.

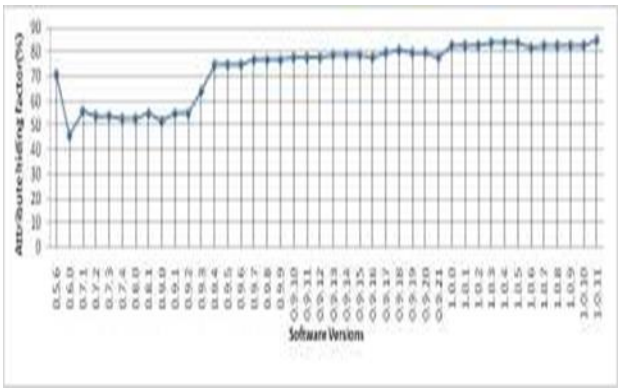

Fig.2: Attribute Hiding Factor (AHF) Metric Trend

2) 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.

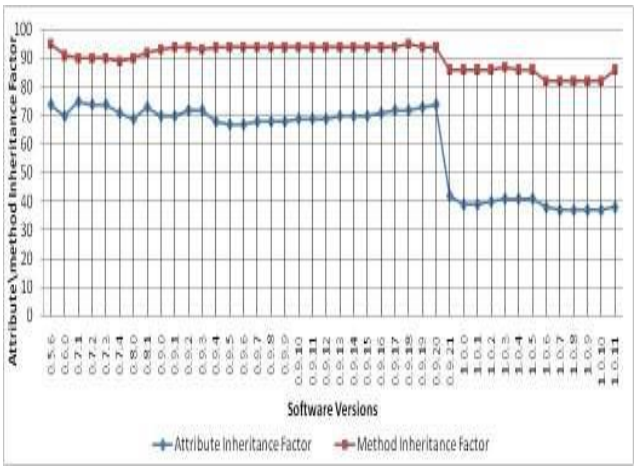

Fig.3: Attribute Inheritance Factor and Method Inheritance Factor Metrics Trends

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. It may be noted that at class level, the metrics related to method inheritance and attribute inheritance show an upward trend towards the latest versions. However, on average number of inherited attributes has been very less in comparison to number of inherited methods. Probably due to this, decline in values of AIF is sharper in comparison to MIF.

3) 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. PF can be calculated as follows:

M (Ci) = Number of New Methods
M (Ci) = Number of Overriding Methods
DC(Ci) = Descendants Count

In successive versions of this software, 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.

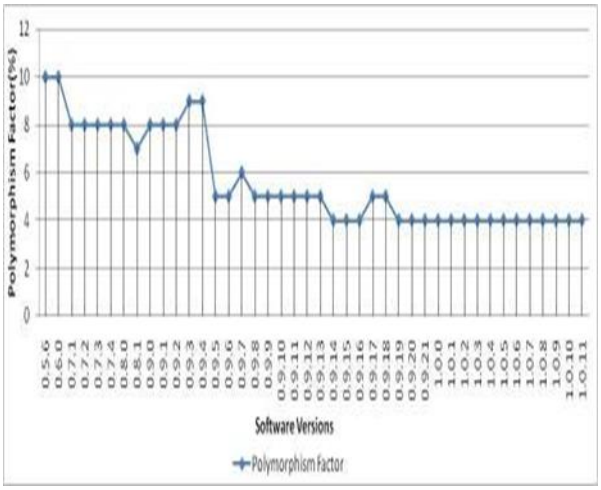

Fig. 4: Polymorphism Factor (PF) Metric Trend

4) Coupling Metrics

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 (Abreu et al., 1996). At package level, Robert Martin defines coupling in two forms: Afferent Coupling (Ca) and Efferent Coupling (Ce) (Martin, 2003). Efferent coupling keeps track of outgoing dependencies to other packages whereas afferent coupling relates to incoming dependencies from other packages. He further defines another metric instability in terms of these two metrics.

Coupling between classes can be inbound (import) or outbound (export). Coupling Between Classes (CBC) measures the number of inter dependencies a class has with other classes in the design. It takes into account all types of associations - incoming as well as outgoing, and strength (number) of interdependencies. In this study, different metrics are selected to measure these different dimensions of coupling. CBCinM measures incoming dependencies from multiple classes. It counts the number of classes which are dependent upon this class. It also considers the individual interdependencies separately (multiple dependencies between same pair of classes are counted separately). CBCinU measures incoming dependencies but counts the multiple interdependencies in any two classes only once. Similarly CBCoutM and CBCoutU measure the outgoing dependencies. EC_Par and IC_ParU measure the export and import coupling with respect to a class's usage as a data type in other classes or use of other classes as a data type in the class.

Another kind of coupling is related to the dependencies a class has on other classes in the same scope (within a package), same scope branch (with classes in other related packages), or not in the same branch (with classes in unrelated packages). Interclass dependencies within the same branch or in the same scope branch are easy to manage than interdependencies with classes not in the same scope branch as the class itself. NumAssEl_sb, NumAssEl_nsb, and NumAssEl_ssc metrics measure these types of couplings.

B. Package Level Metrics

In object oriented terms, a package is a collection of classes. Martin (2003) defined some metrics to evaluate design of packages. This section analyzes the metric results and interprets the values to discuss the trends in package design quality.

1) Relation Cohesion

Figure 5 presents average relation cohesion in packages of the software component across its different releases. Metric values have improved over the passage of time. As classes inside a package should be strongly related, the cohesion should be high. On the other hand, too high values may indicate over coupling. A good range for relation cohesion is 1.5 to 4.02. Assemblies where relation cohesion is <1.5 or >4.0 might be problematic. As per this rule of thumb in this component all releases, except 0.9.21, are problematic.

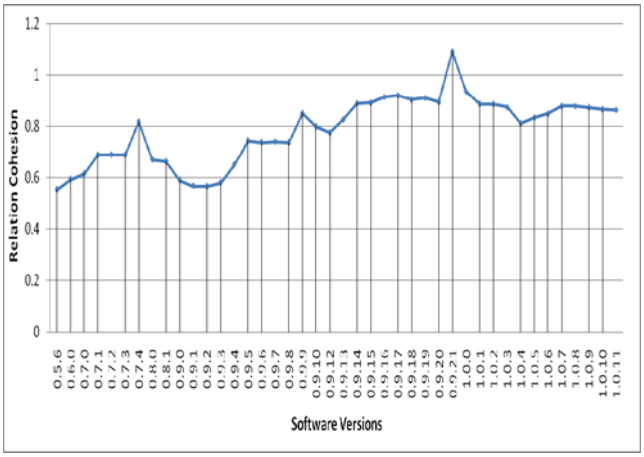

Fig. 5: Relation Cohesion Metric Trend

2) Instability

The instability metric is a normalized metric which combines efferent coupling with afferent coupling and gives instability as ratio of efferent coupling to sum of efferent coupling and afferent coupling. The metric range for this component varies from 0.7 to 0.8 with a few exceptions (Figure 6). This indicates that on average a classes inside a package are dependent upon classes outside the large number of package. As the component evolves, average instability of the packages remains high. It indicates that outside changes will affect the internal design of an average package. Instability measured alone cannot give some useful hints; it should be studied along with the abstractness metric discussed next.

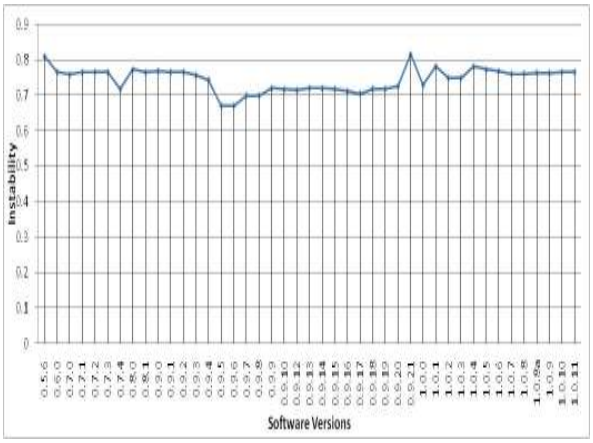

Fig. 6: Instability Metric Trend

3) Abstractness

Instability of packages in all versions of this component has been high throughout. However, at the same time abstraction level of packages has decreased gradually (see Figure 7). It indicates that after several extensions/modifications, the packages have become more concrete as perhaps more number of concrete classes and less number of abstract classes have been added to the packages.

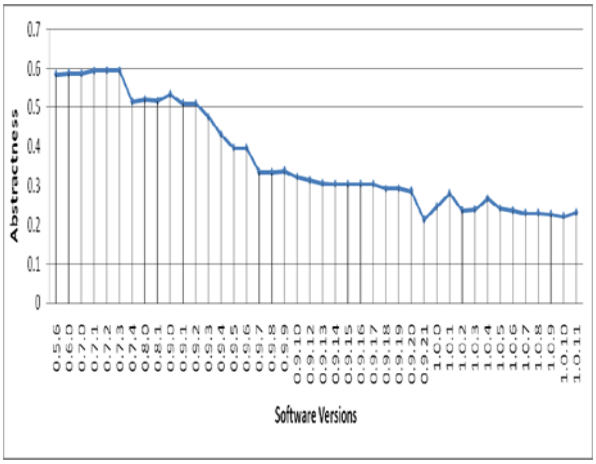

Fig. 7: Average Abstraction Level of Packages.

Packages were having sufficient levels of abstraction in the beginning, which was useful to extend them to support future requirements. Now the component has become general enough to support the maximum possible requirements in its domain. It is supported by the fact that number of feature requests have also reduced with time.

Good package design is achieved by perfect balance between abstraction (A) and instability(I). Another metric in this metric set known as the Distance from the Main Sequence, measures this balancing act.

4) Distance from the Main Sequence

The normalized metric D measures the relation in abstraction (A) and instability (I). The metric takes values nearly zero for a package, if A and I are perfectly balanced. As shown in Figure 8, metric values have improved considerably in successive releases of the software component. Specifically recently after version 1.0.5, it has attained a value nearly zero. This indicates that on average package design is balanced with respect to abstraction and instability.

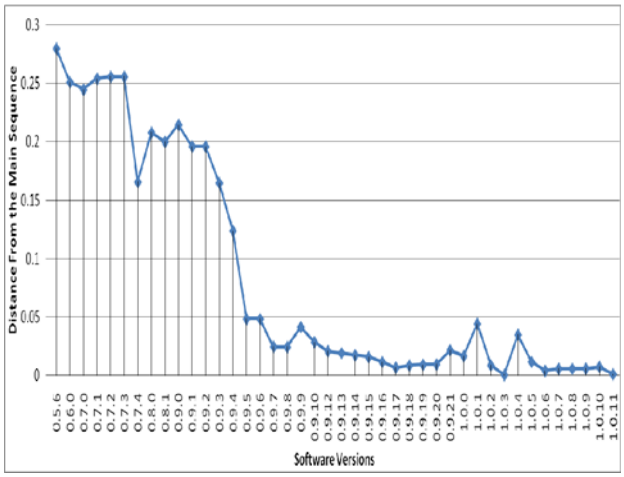

Fig.8: Distance from the Main Sequence Metric Trend

V. Conclusion

This work examines several versions of the JFreeChart software using object oriented metrics at three levels: system, package, and class. Object oriented metrics facilitate to assess the usage of basic conceptions of the object oriented paradigm such as abstraction, polymorphism, inheritance, coupling, and cohesion whilst designing applications derived from this theory. These conceptions of the paradigm are empirically validated to be related to creating reliable, resilient, and easily maintainable designs. Main aim of this work is to examine as to what extent the concepts of object orientation are incorporated in the software design and how the usage of these concepts perks up or degrades as the design evolves over a course of time.

REFERENCES

[1] Demyanova, Yulia, Thomas Pani, Helmut Veith, and Florian Zuleger. "Empirical Software Metrics for Benchmarking of Verification Tools." In Computer Aided Verification, pp. 561-579. Springer International Publishing, 2015.

[2] Padmini, K. V., H. M. N. Dilum Bandara, and Indika Perera. "Use of software metrics in agile software development process." Moratuwa Engineering Research Conference (MERCon), IEEE, 2015.

[3] Ferreira, K. A. M., Bigonha, M. A. S., Bigonha, R. S., Mendes, L. F. O., and Almeida, H. C., "Identifying Thresholds for Object-Oriented Software Metrics", The Journal of Systems and Software, Vol. 85, pp. 244–257, 2012.

[4] Goel, B. M., and Bhatia, P. K., "Analysis of Reusability of Object-Oriented System using CK Metrics", International Journal of Computer Applications (0975 – 8887), USA, Vol. 60, No.10, , pp. 32-36, 2012.

[5] Goel, B. M., and Bhatia, P. K., "Analysis of Reusability of Object-Oriented Systems using Object-Oriented Metrics", ACM SIGSOFT Software Engineering Notes, ACM, New York, USA, Vol. 38, No. 4, pp. 1-5, 2013.

[6] Sastry, B. R., and Saradhi, M. V. V., "Impact of Software Metrics on Object Oriented Software Development Life Cycle", International Journal of Engineering Science and Technology, Vol. 2, No. 2, pp. 67-76, 2010.