Process and Product Metrics to Assess Quality of Software Evolution

Kuljit Kaur
Dept. of Comp Sci & Engg
Guru Nanak Dev University,
Amritsar, India-143005

ABSTRACT
During the evolution of a software system, bugs are found and fixed, and also the system is adapted to meet the new set of requirements. In terms of software evolution, quality is related to the ease of the application to accommodate major changes. The continuing change process deteriorates the design of the system and, further increases the complexity of the software and the system becomes more difficult to evolve. A well designed software product is supposed to incorporate changes easily. So improvements in design structure may also determine the ease with which the software evolves. This paper analyzes the phenomenon with the help of software metrics. It employs process and product metrics to analyze a large software product to assess its internal structure and its ease of evolution. The metrics data indicates that the internal structure of the software product has improved over time and so has the quality of its software evolution..

General Terms
Software Engineering, Software Evolution, Process Metrics, Product Metrics.

Keywords
Software Metrics, Software Evolution, Process Metrics, Product Metrics.

1. INTRODUCTION
Software evolution tasks relate to the work, programming and other activities, performed to produce new versions or releases of existing operational software system. During the evolution of a software system, bugs are found and fixed, and also the system is adapted to meet the new set of requirements. In terms of software evolution, quality is related to the ease of the software system to accommodate major changes. It is a common perception that as a program ages, more and more changes degrade the structure of a software system. The continuing change process deteriorates the design of the system and, further increases the complexity of the software and the system becomes more difficult to evolve. A well designed software product is supposed to incorporate changes easily. So improvements in design structure may also determine the ease with which the software evolves. Software metrics are useful in many ways to create quality

software products within budget and time constraints. They help in project estimation and progress monitoring, evaluation of work products, process improvement, and experimental validation of best practices [1]. Goodman defines software

metrics as [2]: "The continuous application of measurement-based techniques to the software development process and its products to supply meaningful and timely management information, together with the use of those techniques to improve that process and its products". In this paper, the focus is on product and process metrics. Software process and product metrics are quantitative measures that enable software people to gain insight into the efficacy of the software process and the projects that are conducted using the process as a framework [3].

2. PROCESS METRICS
All A software process is the set of activities, methods, and practices that are used in the production and evolution of software. IEEE defines a process as “a sequence of steps performed for a given purpose” [4]. A software process refers to the way in which software projects are carried out, and the methodologies and techniques that are used to develop the required software. Software development life cycle models, such as the Waterfall model, can be used to describe the overall structure of a software process. A software development life cycle model specifies a sequence of phases such as requirements specification, analysis, design, coding, testing, installation and maintenance. But it does not prescribe specific methods or techniques to be used.

Process metrics relate to the software development process, comprising the activities, methods, and standards used. The use of software process metrics has enabled some organizations to much more effectively understand and control their software development process [5]. In this regard, the Capability Maturity Model by Humphrey and others at the Software Engineering Institute have been among the most influential [6]. Process metrics consist of the following:

- Maturity- and defect- metrics
- Management metrics
- Team resourcing and method metrics, e.g., productivity metrics based on effort, cost, milestone dates, duration
- Metrics for measuring project progress
- Metrics for measuring program dynamics, e.g., growth or change requests If any or all of these is increasing, this may indicate a decline in quality of software evolution [3].

This paper analyzes only two metrics from this set of metrics. These two metrics are

- Number of requests for corrective maintenance, and
- Average time taken to implement a change request

3. PRODUCT METRICS
Product metrics relate directly to the result of a software development process. Important features of a software product

that are often measured include but are not limited to: size, quality, user requirements, product growth, and user comfort. Product measures are, for example, as follows:

- Architecture level measures (e.g., number of components, layers, coupling)
- Quality measures (e.g. maintainability, efficiency, reliability, and usability)
- Size related measures – (e.g. LOC, and function metrics)
- Documentation (e.g. Length and readability of the documents)
- Software and System complexity (both structural and data related).

Product metrics can be used to measure a number of different quantities related to the physical software product. Measurements can be made at all stages of the development process, however not all metrics can be applied to all stages. Early estimates of the final product are valuable to management in determining the feasibility of a project. Metrics which can be measured earlier in the development processes are of greater value e.g. it is beneficial to be able to estimate size early in the development process to carry out the cost/benefit analysis of a software development project.

Product metrics are collected at different stages of the software development life cycle such as requirement analysis, design, code, and test. Product metrics can also be collected at different levels of granularity- method, class, package, and system level.

This section gives an account of the system level metrics to measure the structural properties of an object oriented software system in the design phase. 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 [7]. This set has metrics to measure the basic properties of an object oriented design such as information hiding, 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 [8]. 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.

Metrics, categorized according to the elements of the object oriented paradigm they measure, are given below:

3.1 Information Hiding

Method Hiding Factor (MHF) and Attribute Hiding Factor (AHF) measure the degree of information hiding in a system [7].

3.2 Inheritance

Method Inheritance factor (MIF) and Attribute Inheritance Factor (AIF) are the system level metrics that measure the usage of inheritance mechanism [7].

3.3 Coupling

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 [7].

3.4 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 [7] measures the polymorphic behaviour of classes taken together.

4. METRICS AND QUALITY OF SOFTWARE EVOLUTION

As a software product evolves, it is likely to get changed. Most software systems evolve over a number of releases, each new release involving the following activities: (i) defect identification/repair, (ii) addition of new functionality, (iii) removal of some existing functionality, and (iv) optimizations/improvements. A better understanding of the way a software product evolves during its lifetime may provide useful information for designing new products. In the context of software evolution, the following classes of metrics have been identified: effort, activity, control, and environment [9, 10].

Brief descriptions of the metrics are given below:

• Effort reflects the amount of resources (e.g., number of person-hours per month) applied to evolution.

• Activity represents the amount of work (e.g., size of system, functionality, number of changes to existing functionality) accomplished over the interval.

• Control is related to the metrics which reflect factors that, under direct management control, are aimed at controlling evolution process outcome (cost, quality, interval).

• Environment is related to the metrics which reflect factors that, outside direct management control, influence evolution process outcome (cost, quality, interval).

5. EXPERIMENTAL ANALYSIS

This study is based on the data from a large open source project available at one of the largest storehouses of open source projects www.sourceforge.net. The software under study is a charting library, known as JFreeChart that developers can use to display professional quality charts in their applications. This research studies over 43 versions of this project released in the time period from 2000 to 2008. The information regarding JFreeChart's bug reports is presented in Table 1 for easy analysis.

Bug reports information is taken year wise as version wise information was not available. Number of bugs reported is following a downward trend after year 2005. Reduction or absence of bugs is also related to increasing software quality [11]. A decreasing number of bug reports support the fact that quality of this software product has improved overtime.

Table 1: Bug reports for the Software Product during its Evolution.

Year	Bug Reports
2001	15
2002	71
2003	219
2004	100
2005	164
2006	118
2007	111
2008	79

5.1 PRODUCT METRICS

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

1. Method Hiding Factor (*MHF*) and Attribute Hiding Factor (*AHF*)

MHF and *AHF* , *together*, represent the use of the information hiding mechanism 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. 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 [12]. It could be observed from Figure 1, that the *MHF* metric remains within the prescribed limits for all the releases of the software component. On the other hand in Figure 2, 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 product adheres to the concept of information hiding.

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. 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.

3. Polymorphism Factor (PF)

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 product (figure 4), 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 [13].

4. Coupling Factor (CF)

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.

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 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.

5.2 PROCESS METRICS

An analysis of the bug-fix requests and the time taken to fix them is expected to give some insights into the quality of the software evolution. Every bug-fixing activity results in a few or more changes in the software product. Changes in a modularized software product are expected to be local and hence effort required to carry out the change is supposed to be less. As it has been observed that the structure of the software is modularized over the period of time, so bug resolution time should also reduce expectedly.

The bug tracking system for the software component accepts bug reports from the users. It is noticed that some bugs are reported to the key developer of the product (username-mungady) via email, who then records these bugs in the bug repository using the bug tracking system. When a bug is submitted to the bug tracker, its status is 'open'. Once it is fixed, its status is changed to 'closed'. This study filters the bug repository to extract information about the bugs having the status as 'closed' and resolution as 'fixed'. Other possible resolutions are: invalid, when the problem described is not a bug; won't fix, when the problem described is a bug which will never be fixed; duplicate, when the problem is a duplicate of an existing bug; and works for me, when there is no test case against which the bug can be reproduced. Every bug is not attended to as soon as it is submitted. Sometimes, a lot of bug-fixing activity happens near an important release. Bug reports are perhaps given more consideration while preparing new releases. Users submitting the bugs can also submit the solutions. Such solutions are tested and if found to be correct are committed to the version log. Otherwise, bug report is assigned to one of the developers. He resolves the related issue and commits the solution to the repository. However, it is not possible to determine the actual effort spent by an assignee on a bug. So bug life time is defined as the difference between submission date and closing date of the bug. We intend to study the effect of modularization on the bug life time. It is assumed that bug life time will decrease as improvements in design structure should localize the changes and reduce the effort to make changes.

The data set available at the bug tracking web page of the project contains information about 1024 bugs which has been submitted in the bug-repository of the project. The number of bugs in the bug repository with the status as 'closed' is 524. For the JFreeChart software component, Figure 6 plots the number of bug-fix requests which are opened in a particular

[1] http://www.aivosto.com/project/help/pm-oo-mood.html#abreumelo

year, and also the bug-fix requests which have been closed out of the set of open requests. There seems to be a lot of difference in the number of open and closed bug-fix requests, but closed bug-fixes here are the ones with resolution as 'fixed'. Bug-fixes with resolution as 'invalid' and 'duplicate' etc. are not considered here. Maximum activity is in the year 2003. A total of 22 versions are released in years 2002 and 2003 with 11 versions released in each year. Another peak in the activity occurs in the year 2005 when the stable version of the software component JFreeChart 1.0.0 is released. After this the number of bug-fix open requests as well as the number of closed bugs decrease.

Next important consideration is to see how the bug life time changes over the period of time. Change in bug fix life time as the software component evolves is given in figure 7. It shows that the bug fix life time improves significantly over the period of time. It is interesting to see that highest peak in bug life time corresponds to the initial versions of the software component. It could be observed that bug fix life time has decreased drastically from the year 2001 to year 2002. It remained constant in the years 2002 and 2003 (45 days).

6. CONCLUSIONS

Software metrics play an important role in the software development life cycle. A number of metrics have been proposed and studied in this context. However, the set of metrics, for analysis of the state of a software product once it is deployed, are very less. This paper studies the metrics in this category. It compares the metrics to evaluate quality of software evolution from two different perspectives i.e. process and product. It has been observed in the experimental analysis of a large software product that a software product with good characteristics evolves with ease. The requests for its corrective maintenance go down with time and at the same time the time to make corrections decreases.

7. REFERENCES

[1] Grady, R. (1994).Successfully Applying Software Metrics, *IEEE Computer* 27(9): 18-25.

[2] Goodman, Paul. (1993). *Practical Implementation of Software Metrics.* London: McGraw Hill.

[3] Sommerville, I. (2000). Software Engineering, Addison-Wesley, 6th Edition.

[4] IEEE Std 610.12-1990: IEEE standard glossary of software engineering terminology, 1990.

[5] Pfleeger, S.L. and C.L. McGowan (1990), "Software Metrics in a Process Maturity Framework," *Journal of Systems and Software*, July, 255–261.

[6] Humphrey, W. (1988). Characterizing the software process: a maturity framework, IEEE Software 5 (2): 73–79.

[7] 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.

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

[9] Rami1, J. F. and Lehman, M. (2000). Metrics of Software Evolution as Effort Predictors - A Case Study, IEEE, 2000.

[10] Rami1, J. F. and Lehman, M. (2001). Defining and Applying Metrics in the Context of Continuing Software Evolution, IEEE, 2001.

[11] Murgia, A., Concas, G., Pinna, S., Tonelli, R. and Turnu, I. (2009). Empirical Study of Software Quality Evolution in Open Source Projects Using Agile Practices, Computing Research Repository *(CoRR)*, arXiv.org e-Print 0905.3287.

[12] 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.

[13] 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.

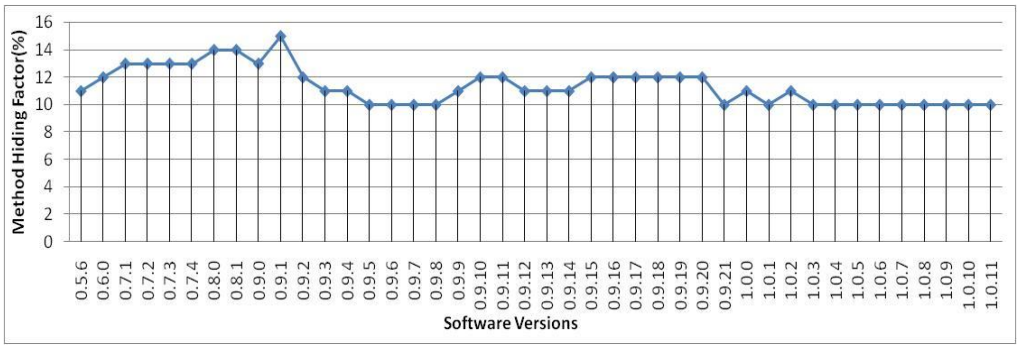

Fig.1: Method Hiding Factor (*MHF*) Metric Trend

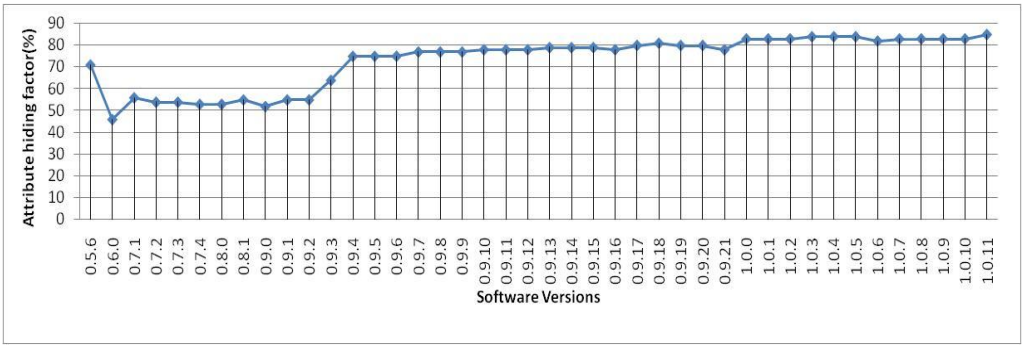

Fig.2: Attribute Hiding Factor (*AHF*) Metric Trend

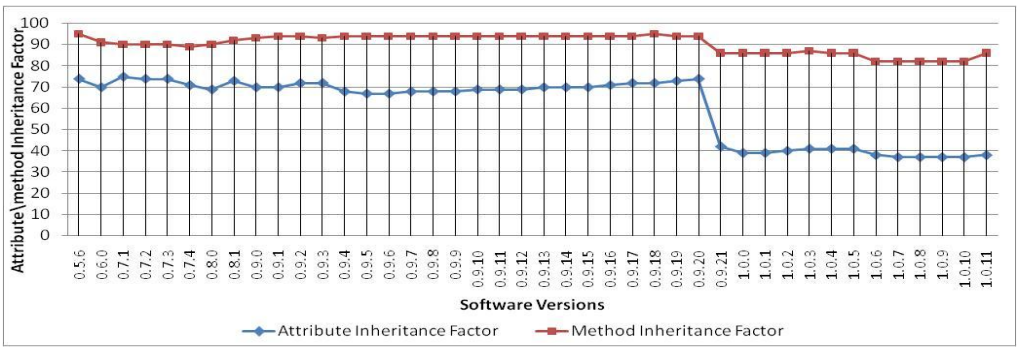

Fig.3: Attribute Inheritance Factor and Method Inheritance Factor Metrics Trends

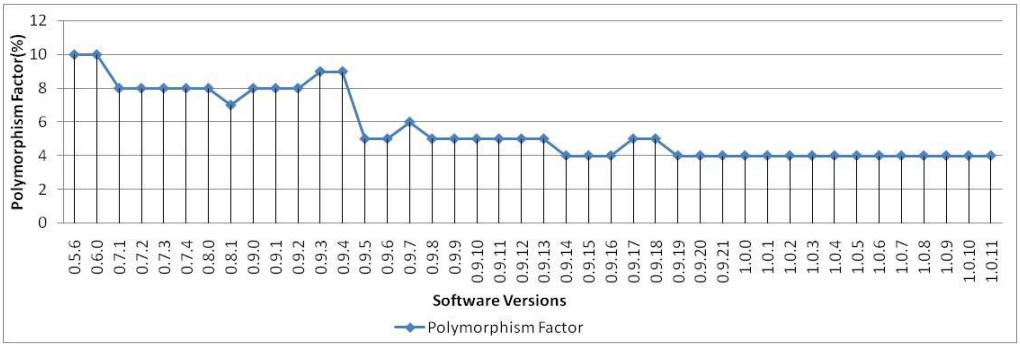

Fig.4: Polymorphism Factor (*PF*) Metric Trend

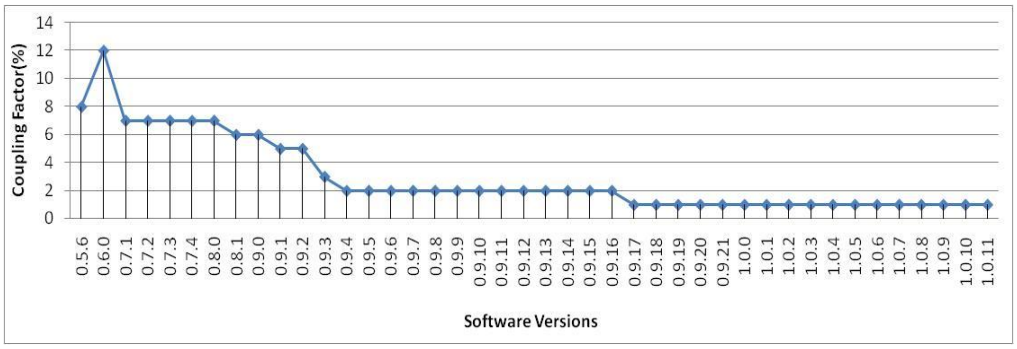

Fig.5: Coupling Factor (*CF*) Metric Trend

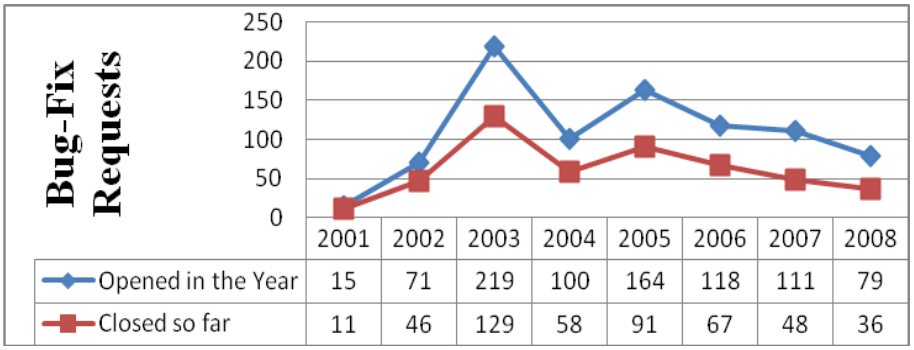

Fig.6: Year wise analysis of bug-fix requests.

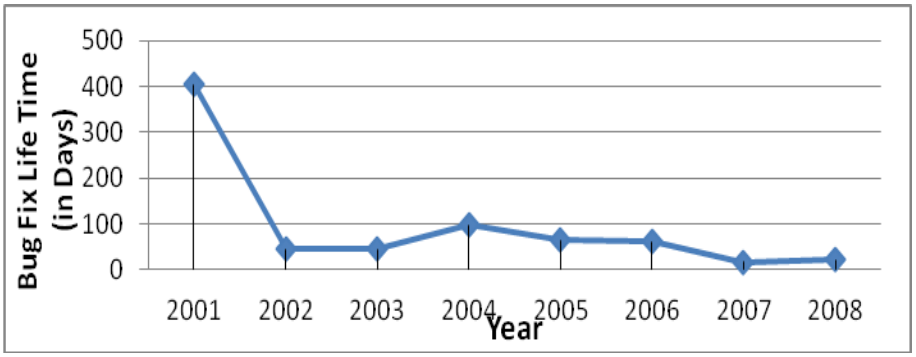

Fig.7: Change in bug fix life time over the period of time.