Effect of software evolution on software metrics: An open source case study

Kalpana Johari
Centre for Development of Advanced Computing
CDAC, NOIDA
NOIDA, India
kalpanajohari@cdacnoida.in

Arvinder Kaur
Guru Gobind Singh Indraprastha University
Sector-16 Dwarka
Delhi, India
arvinderkaurtakkar@yahoo.com

ABSTRACT

Software needs to evolve in order to be used for a longer period. The changes corresponding to corrective, preventive, adaptive and perfective maintenance leads to software evolution. In this paper we are presenting the results of study conducted on 13 versions of JHot Draw and 16 versions of Rhino released over the period of 10 years. We measured Object Oriented Metrics and studied the changes in the measured values over different releases of two medium sized software developed using Java. We also investigated the applicability of Lehman's Law of Software Evolution on Object Oriented Software Systems using different measures. We found that Lehman's laws related with increasing complexity and continuous growth are supported by the data and computed metrics measure.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Software Evolution

General Terms

Verification.

Keywords

Software evolution, laws of software evolution, software metrics, revisions, versions, software complexity, open source.

1. INTRODUCTION

Software is not prone to wear and tear but still it may become useless if not revised in response to ever changing user requirements. Software needs to evolve in order to be used for longer period. Lehman et al has done extensive research on evolution of large and long lived software [1]. Lehman's laws of software evolution, based on the empirical study, indicate that continuous change and growth is required for keeping the software long-lived. The laws also suggest that over the period, due to changes and growth, software system becomes complex and it becomes more and more difficult to add new functionalities to it. It has been more than two decades since laws being proposed but still there are very few empirical studies to support the applicability of laws to software systems and sub systems more so in the case of object oriented software system.

The availability of source code and change details of open source software has provided a support to the study of software evolution. The work presented in this paper is based on the study of several versions of two open source software i.e. JhotDraw and Rhino. Both the software used for study have been developed using Java. The main objective of the study is to examine the applicability of Lehman's laws to object oriented software system. We have computed the object oriented metrics, proposed by Chidamber and Kemerer[2] , for the two software. The computed metrics value for different releases has been used as the basis for examining the laws of software evolution.

The paper is organized as follows: Section 2 provides the background, giving brief interpretation of Laws of software evolution and object oriented metrics. Section 3 contains the brief introduction to the two case studies. Section 4 explains the approach and data representation. Section 5 presents the analysis and interpretation of laws with respect to object oriented software. Section 6 presents the related work, followed by conclusion and future work.

2. BACKGRUOND

This section consists of two sub section. The first one gives a brief interpretation of Lehman's Laws of Software evolution and the second one presents the meaning of object oriented metrics used in the study

2.1 Lehman's law of Software Evolution[1]

- Continuing Change (1974): E-type systems should continuously be changed in order to be used for longer period. The required change may be called for in response to change in environment, as a bug fix exercise or as a preventive maintenance activity or any other activity leading to change.
- Increasing Complexity (1974): The complexity of an E-type system increases unless some preventive maintenance is done to control it. Increase in complexity may arise due to number of changes or due to addition of more functionalities leading to more interaction.
- Self Regulation (1974): Evolution process of E-type system is self regulatory. This means that growth rate of is regulated by the maintenance process. There is a balance between what is desired to be changed and what can actually be achieved. In order to have a smooth evolution process, the limitations on growth rate should be accepted.
- Conservation of Organizational Stability (invariant work rate) (1980): Evolution process of software conserves the organizational stability. The work rate of an organization evolving a large software tends to remain constant. This means it is hard to change the staff who has been working on evolving software. The average global effective rate in evolving software tends to remain constant over product lifetime.
- Conservation of Familiarity (1980): The familiarity with evolving E-type software is conserved. A huge change that might cause lack of familiarity of staff members involved with the evolving software is avoided. For small changes the familiarity of software is easily achieved by the personnel involved with the software. Hence the average incremental growth remains constant as the software evolves.
- Continuing Growth (1980): The functional content of E-type systems must be continually enhanced in response to user feature request in order to maintain user satisfaction over its life period.
- Declining Quality (1996): The Evolution process causes decline in the quality of evolving software.

2.2 Metrics used in the study

We have applied package level, class level and method level metrics on open source software. This section presents the definition metrics used in the study. Although major emphasis has been on object oriented metrics as proposed by Chidamber and Kemerer [2], we have also considered Line of Code (LOC), number of Classes, number of packages, number of Method (NOM) in a class, number of attributes in a class, method line of code (MLOC) as variant of size metrics. For measuring the complexity at different level, we have considered afferent and efferent coupling at package level, coupling between object(CBO) and response for class(RFC) at class level and McCabe's cyclomatic complexity[8] at method level.

- CBO - Coupling between object classes
  The CBO metric for a class is the count of all those classes with which the given class is coupled. Two classes may be coupled due to method call, arguments, return type, field access, inheritance and exception.
- LCOM - Lack of cohesion in methods
  A LCOM metric of a class is the count of set of methods in a class that are disjoint with respect members of a class being accessed by them. The original definition of this metric as presented in [2] considers all pairs of a class's methods.
- NOC - Number of Children
  NOC metrics measure the number of direct descendents of a class.
- RFC- Response for a Class
  The RFC metrics of a class is the measure of number of methods that can be invoked in response to a message received by an object of the class. Ideally RFC should measure the transitive closure of the call graph for each method.
- WMC - Weighted methods per class
  WMC metric for a class is the sum of complexities of its methods. The complexity of an individual method can be measured as cyclomatic complexity or simply we can assign 1 as the complexity value . We have used cyclomatic complexity variant of WMC (WMC using CC).
- LOC-Line of code
  LOC include all the lines of source code, except blank and comments
- NOM:- Number of Methods
  NOM for a class defines the number of method defined in the class. The count of static method is defined using NSM and the number of overridden methods is defined using NORM.
- NOA:-Number of Attributes
  NOA for a class is the number of attributes in that class. The count of static attributes of a class is represented as NSA
- NPM: Number of public methods
  NPM of a class is the number of public methods in a class. The count is included in number of methods.

3. CASE STUDIES

This section presents a brief introduction to the software used in the study namely JhotDraw and Rhino. According to cook the software can be categorized as E-type [4][11]. The software has been implemented in Java and are open source. All the data related to revisions is available on internet. The choice of software is mainly guided by the limitations of metrics measuring tools and the availability of revision details and source code.

3.1 Case study 1: JhotDraw[9]

JhotDraw is a framework for 2-dimensional drawing editors and for document-oriented applications. It is implemented in java. It is an adaption of HotDraw. Since the registration of Jhot Draw 5.2 on sourceforge[9] on 10/10/2000, there has been 12 more official releases till date for which source is available. Of the 12 releases 3 are beta versions namely JhotDraw 5.41b, 5.42b and 6.0b. We have considered 10/10/2000 as the initial release date of software. JhotDraw7 is a result of major revisions applied to previous versions of JhotDraw. It was developed by Werner Randelshofer. So far about 7 different versions of JhotDraw7 have been released. The latest version is JhotDraw7.6.1. Each release of Jhot Draw is a result of set of revisions made to it in response to bug fixation, feature request, adaption to new environment or preventive activity. Since its first release in 2000 that consisted of 9419 LOC, there has been 8 fold increases in LOC till date. The software has gone through 651 revisions till date. Table I gives the details of releases of JhotDraw over the period.

Table 1. Details of releases of JhotDraw

Versions	Release date	LOC	no. of classes	Total NOM	Total Attributes
7.6	06-01-2011	80169	672	5885	1606
7.5.1	01-08-2010	79275	669	5845	1599
7.4.1	16-01-2010	72933	639	5582	1455
7.3.1	18-10-2009	73361	638	5627	1516
7.2	May-09	71675	621	5486	1479
7.1	Mar-08	53753	463	4285	1087
7.0.9	21-06-2007	52913	487	4234	1090
7.0.8	10-01-2007	39179	343	3256	782
6.0 b1	09-Jan-04	21091	301	2809	521
5.4 b2	01-05-2003	21091	301	2809	521
5.4 b1	21-08-2002	20594	296	2723	504
5.3	09-02-2002	14611	208	1896	380
5.2	10-10-2000	9419	148	1229	271

3.2 Case Study 2: Rhino[10]

Rhino is a JavaScript/ECMAScript interpreter and compiler. It is and open source written entirely in java. Rhino began life as an industrial project at Netscape and was then transitioned to open source. It is being managed by Mozilla foundation. Since its initial release in 1999, there have been 16 releases so far. The first release, Rhino1_4R3, consisted of 20335 LOC and the latest release consists of 43425 LOC. The size in terms of LOC has almost doubled from initial release to current release. Table 2 presents details of releases of Rhino.

4. METHODOLOGY USED FOR DATA COLLECTION

In order to conduct the study with the said objective we required the source code, measure of several metrics and revision details. Both the software used in the study are open source software therefore, the availability of source code for different versions was never a problem. The computation of different metrics was done using various tools meant to compute metrics The average measure of various metrics for releases of JhotDraw and Rhino is presented in Table III and IV respectively. The revision details of each release of software are maintained in the subversion repositories. The revision details of JhotDraw is available at [12] and that of Rhino is available at [13].

5. ANALYSIS OF EVOLUTION OF JHOTDRAW AND RHINO

In this section we analyze the Lehman's laws of software evolution in the light of measures of several metrics, computed for different versions of JhotDraw and Rhino released through evolution process. Some of the laws have a direct relevance to the computed metrics whereas for some of the laws we did not find direct relevance to software metrics

Table 2. Details of Release of Rhino

Versions	Release Date	LOC	no. of classes	Total NOM	Total Attributes
Rhino 1.7R2	22-03-2009	43425	146	1561	632
Rhino 1.7R1	06-03-2008	42830	145	1541	627
Rhino 1.6R7	20-08-2007	39930	133	1434	579
Rhino 1.6R6	30-07-2007	39914	133	1433	579
Rhino 1.6R5	19-11-2006	37960	123	1369	573
Rhino 1.6R4	10-09-2006	37960	123	1369	573
Rhino 1.6R3	24-07-2006	37946	123	1368	573
Rhino 1.6R2	19-09-2005	37771	123	1360	572
Rhino 1.6R1	29-11-2004	37961	126	1366	570
Rhino 1.5R5	25-03-2004	36051	127	1323	559
Rhino 1.5R4.1	21-04-2003	33800	123	1364	518
Rhino 1.5R4	10-02-2003	33718	123	1358	517
Rhino 1.5R3	27-01-2002	32421	113	1281	484
Rhino 1.5R2	27-07-2001	32060	113	1255	480
Rhino 1.5R1	10-09-2000	29495	110	1053	446
Rhino 1.4R3	10-05-1999	20335	84	802	318

5.1 Law 6: Continuing Growth

According to this law the functionality provided by the software should continually grow so as to provide user satisfaction over longer period. Growth can be interpreted as increase in the size of code or increase in the functionality being provided by the software. Whether software has grown in size can be determined by observing the variants of size metrics over subsequent releases. Similar approach was used by Lehman. We computed and compared the LOC, number of classes, Number of methods, Method Line of code for different releases of JhotDraw and Rhino.

Growth of evolving software, in terms of functionality, can be measured by observing change in the number of classes, number of methods and number of public methods. The interpretation of continuing growth as functional growth was also used in [12].

5.1.1 Size Metrics

The graphs in figure 1 and figure 2 show the growth in line of code for JhotDraw and Rhino. All the observations were made with respect to release date and not with respect to versions. Lines of code for the given case study include each line of code except blank and comments. The growth of LOC for the JhotDraw over different releases was observed to be liner where as the growth of LOC for Rhino seemed sublinear. The growth curves of number of classes and NOM, as given in figure 3& figure 4, figure 5 & figure 6, for both the software, were almost similar to growth curve for LOC. Considering the growth curve according to size metrics, linear to sub linear growth rate was observed, there by suggesting that the law of continuing growth is reflected by Object oriented software. We found our observations to be similar to the one made in [14].

5.1.2 Function Metrics

Function Metrics: The growth of software over several releases in terms of functionality can be attributes to increase in number of methods, number of class and number of public methods (NPM). For object oriented software an added functionality is provided by adding a class or by adding a public method to a class, there by leading to increase in number of classes and NPM. The growth curve in figure 7 and figure 8 represents the growth of NPM over several releases of JHotDraw and Rhino respectively. The analysis with respect to number of classes and number of methods has been presented with size metrics. The growth curve of NPM for JhotDraw and Rhino shows a linear growth rate for JhotDraw but the growth rate for Rhino is linear during initial period followed by a long phase of sub linear growth rate.

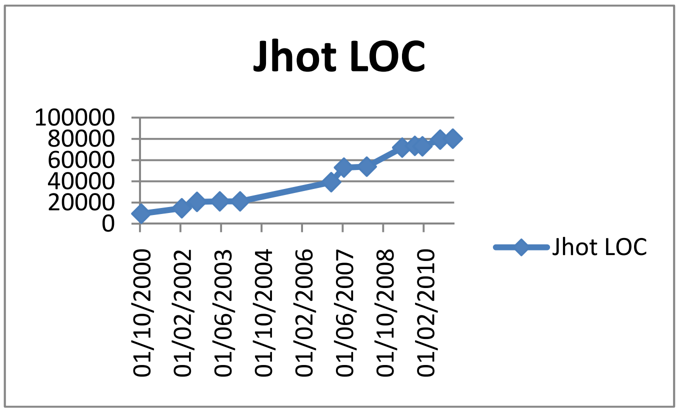

Figure1. Growth curve of LOC for JhotDraw.

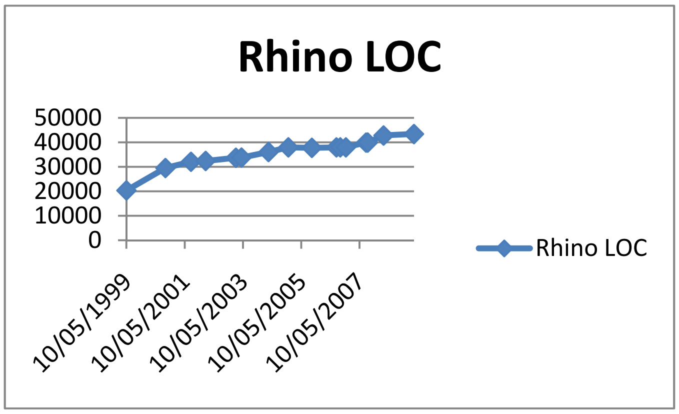

Figure2. Growth curve of LOC for varions releases of Rhino.

5.2 Law 2 Increasing Complexity

According to this law the complexity of software tends to increase over several releases unless some measures are taken to keep the complexity under check. It is generally perceived that growth adds to increase in complexity but if the appropriates changes are made; the evolution process might not show the attributes that conforms increase in complexity. Therefore it becomes difficult to determine whether the law is reflected by the evolution process or not.

In case of object oriented systems, the complexity of the software system or subsystem can be determined using coupling between the classes (CBO), response for a class(RFC) and weighted method per class(WMC). The measures of CBO, RFC and WMC as complexity measures for object oriented software system has been validated through empirical research [15][16]. The measure of McCabe's cyclomatic complexity for member functions does not provide the true complexity measure for object oriented software.

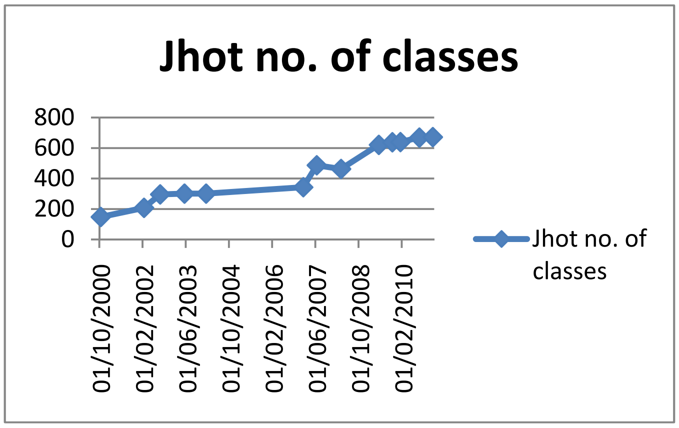

Figure3. Growth curve of number of classes for JhotDraw.

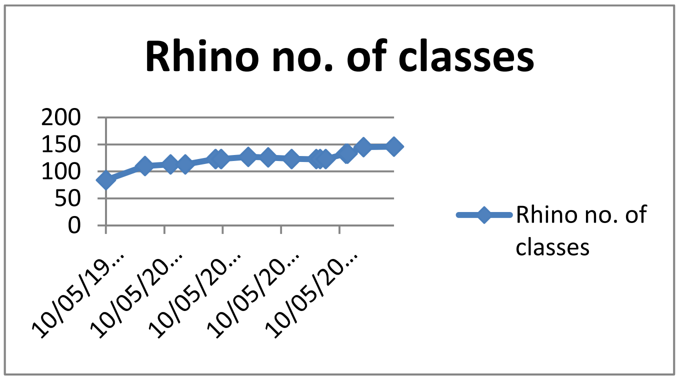

Figure 4. Growth curve of number of classes for Rhino.

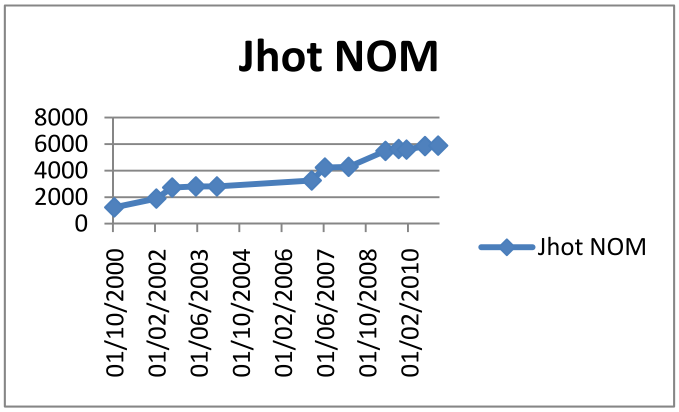

Figure 5. Growth curve of number of methods in versions of JhotDraw.

We computed WMC, CBO and RFC as complexity measure for JhotDraw and Rhino. The change in the measure of total WMC for JhotDraw and Rhino is given in figure 9 and figure 10 respectively. Both the graph shows an increasing trend in complexity measure. For JhotDraw, there was a linear increase observed after the release of JhotDraw 6.0. On looking into the details of revisions made to JhotDraw 6.0 it was observed that a major changes were brought about in the overall structure of the software. The whole software was modified to take advantage of the Java SE 6 platform. For Rhino, the increase in complexity in terms of increase in WMC was observed. The trends were similar for CBO and RFC, given in figure 11 (Jhot CBO), figure12 (Rhino CBO), figure 13 (Jhot RFC) and figure 14 (Rhino RFC) for both the software.

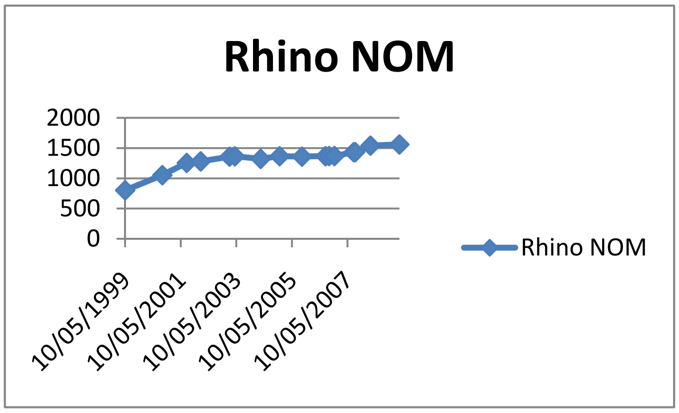

Figure 6. Growth curve of number of methods for various versions of Rhino.

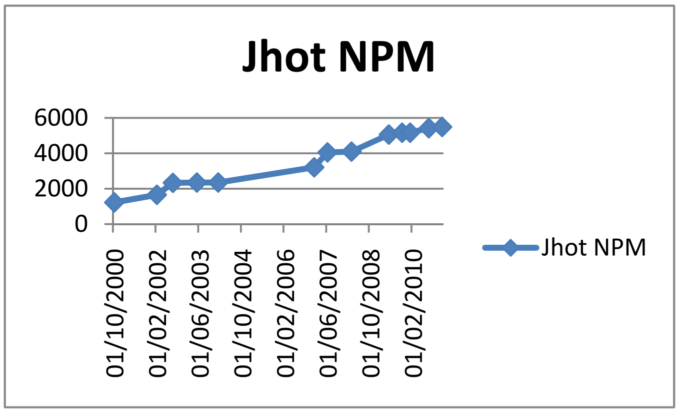

Figure 7. Growth curve of number of public methods in different releases of JhotDraw

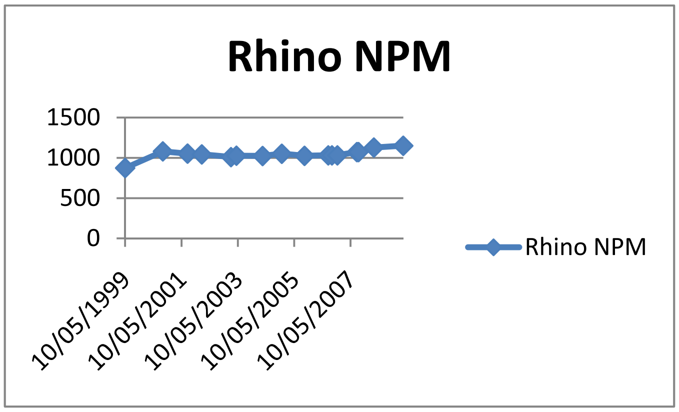

Figure 8. Growth curve of number number of public methods in various releases of Rhino.

On the other hand, if we observe the change in the average measure of WMC for JhotDraw and Rhino given in figure 15 and figure 16 respectively, the increase in complexity of the first version and the latest version is not very substantial. For Rhino, the average WMC increased during initial releases but a downward trend in complexity was observed during the recent releases. In case of JhotDraw, the average WMC increased during the initial releases, but again a decrease in average WMC was observed in the recent releases. The observations for average CBO and average RFC for JhotDraw and Rhino were almost similar to the one observed for WMC and are depicted in figure 17, figure 18, figure 19 and figure 20 respectively.

The interpretation of observation regarding total WMC, CBO and RFC is as follows. The continuing growth of the software led to an increase in the number of classes and the number of member functions thereby causing an increase to the total WMC, RFC and CBO. This indicates an increasing complexity. The interpretation of observations related to average WMC, CBO and RFC is slightly different. Due to increase in the number of classes the average of WMC, RFC and CBO was distributed over to added classes thereby showing a slow increase in complexity. The decrease in the complexity measure for recent releases of Rhino and JhotDraw can be attributed to activities meant to reduce the complexity. Similar observations were also made in [14]. Considering the observations we can consider the law to be reflected by JhotDraw and Rhino.

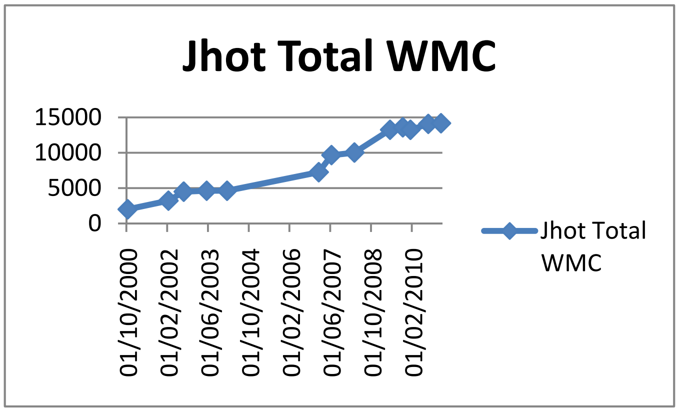

Figure 9. Growth curve of sum of weighted method per class for releases of JhotDraw.

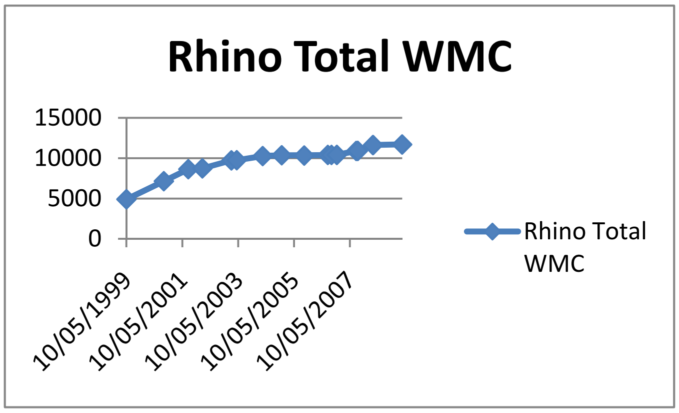

Figure 10. Growth curve of sum of weighted method per class for releases of Rhino.

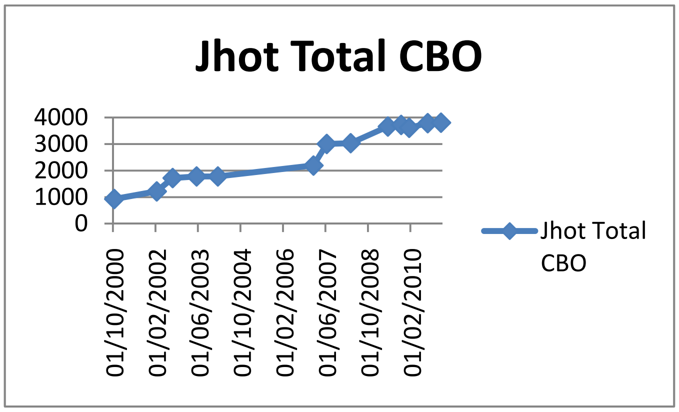

Figure 11. Growth curve for total coupling between objects for various JhotDraw releases.

5.3 Law1 Continuing Change

According to this law, evolving software has to adapt to the changing environment i.e. the software has to continually change in order to be used for longer period. Software may change in response to bug fixing activity or in response to change in the environment. It is difficult to differentiate between growth and change. The change as a result of change in usage environment may include addition of some functions or classes which will increase the size of software leading to growth.

In case of JhotDraw, the version 5.4b2 and 6.0 are absolutely same in terms of LOC and functionality. The overall package structure of 5.4b2 was changed resulting in the release of JhotDraw6.0. Moreover JhotDraw was completely reworked to take the advantage of Java SE 6 platform . In case of Rhino changes were made in response to change in the constructs of Java script. Looking at the observations related to revisions made to JhotDraw and Rhino, we can comment that, the changes provided the longer usable life to the software.

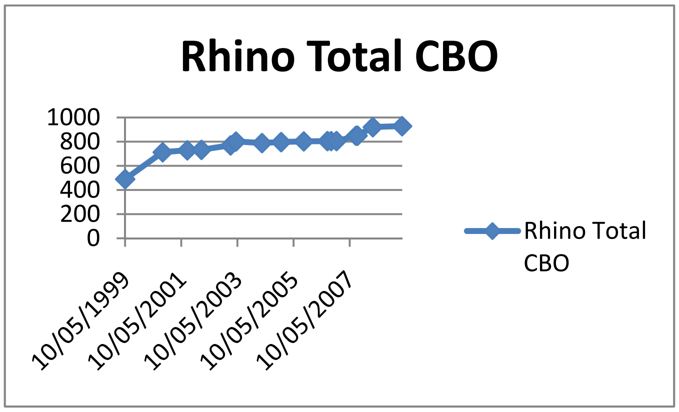

Figure 12. Growth curve for total coupling between objects for various Rhino releases.

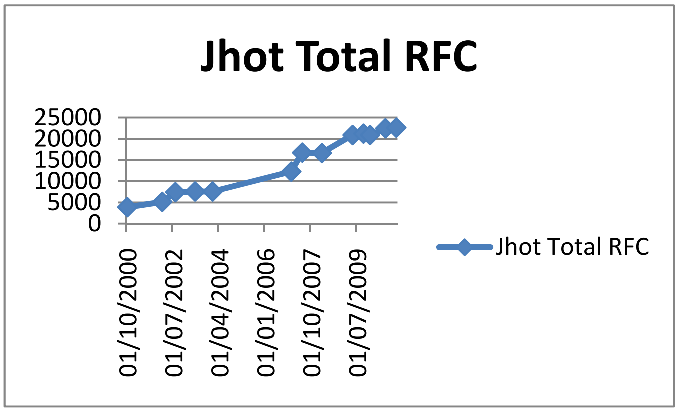

Figure 13. Growth curve for total response for all the classes for different versions of JhotDraw.

5.4 Law 7 Declining Quality

According to this law, the quality of evolving software will decline unless some substantial efforts are made to improve it. The law is somewhat similar to Law 2. Increase in complexity itself is an indicator of declining quality. In case of JhotDraw and Rhino, the law is clearly reflected in the line of law 2.

5.5 Law 8 Feedback system

According to this law, the evolution process is a multi–level, multi agent system. For open source software the law seems to be true since feature request and reporting of bug comes from user community. The existence of feedback system is true for JhotDraw and Rhino but this being multi-level involving multi-agent is hard to determine.

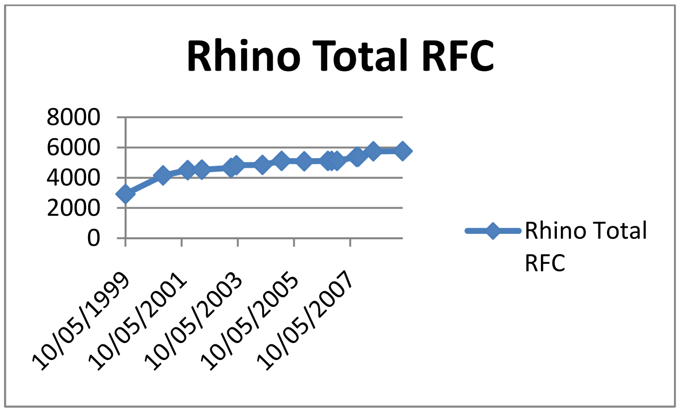

Figure 14. Growth curve for total response for all the classes for different versions of Rhino.

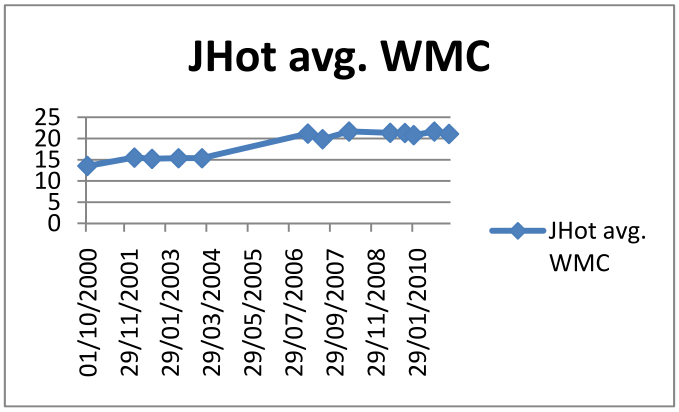

Figure 15. Growth curve for average weighted method per class for different releases of JhotDraw.

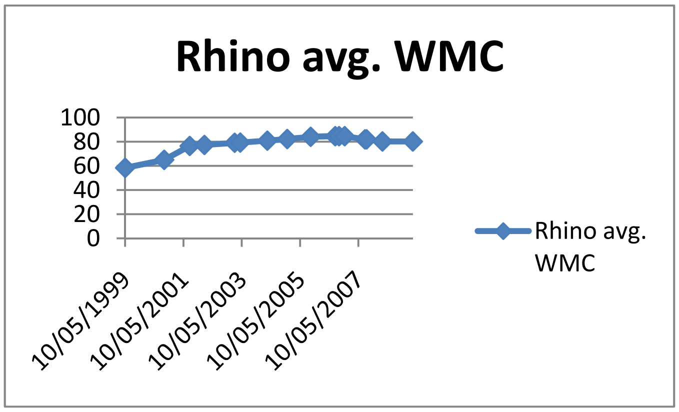

Figure 16. Growth curve for average of weighted method per class for releases of Rhino.

5.6 Law 4 Conservation of organizational stability

According to this law, the average global rate of activity on an evolving system is invariant over the product life time. Determining the average global rate of activity for open source software is extremely difficult if not impossible. The overall process that results in development of software involves community effort and this community generally grows for open source software. Eclipse, Linux, Firefox, Rhino are few examples.

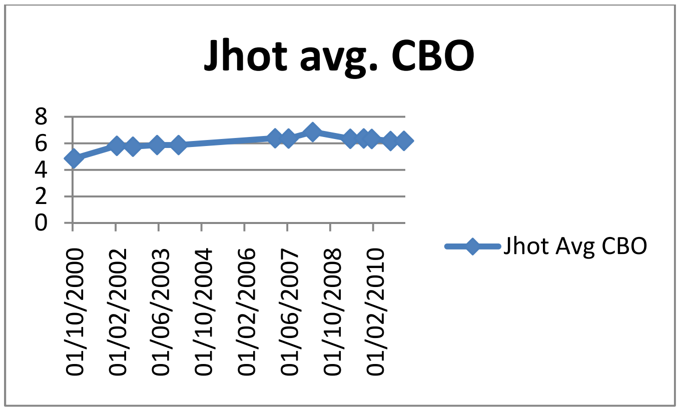

Figure 17. Growth curve for average coupling between the objects for different releases of JhotDraw.

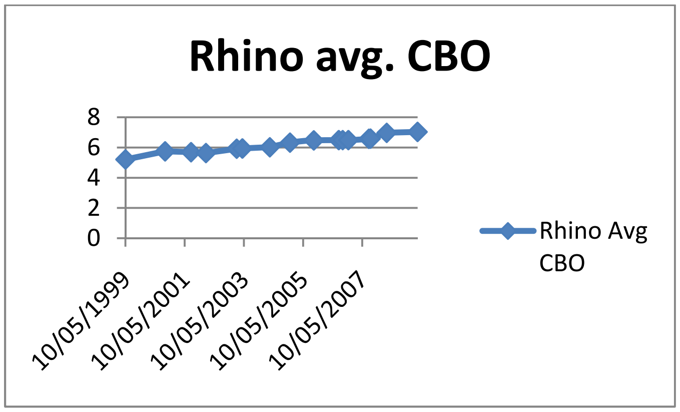

Figure 18. Graph showing growth curve of average coupling between objects for different releases of Rhino.

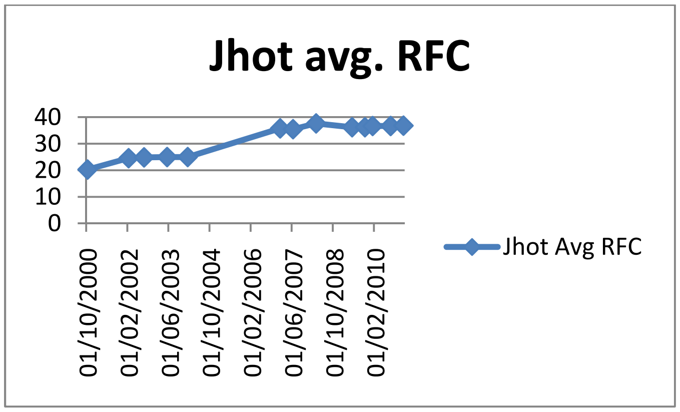

Figure 19. Graph showing growth rate of average response per class for JhotDraw releases.

5.7 Law 5 Conservation of Familiarity

According to this law the changes made to an evolving software during successive releases is limited. This allows conserving familiarity for developer and maintainers. By the study of revisions of JhotDraw and Rhino we observed that the number of source file/ class files involved in single revision were few. There was only one instance where whole of the package structure of Jhotdraw was changed but the change was similar to all the source files and therefore familiarity was maintained.

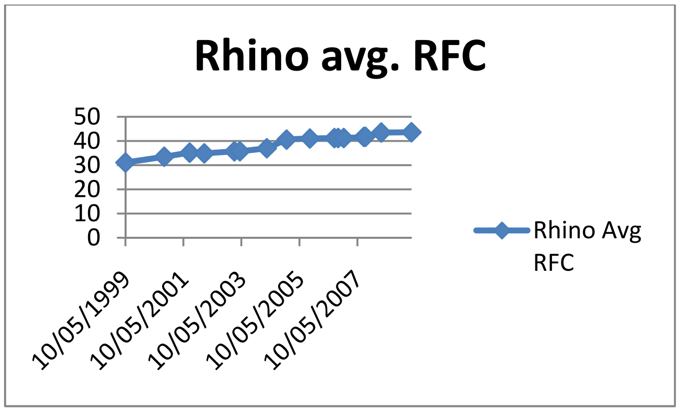

Figure 20. Graph showing growth curve of average response per class for different releases of Rhino.

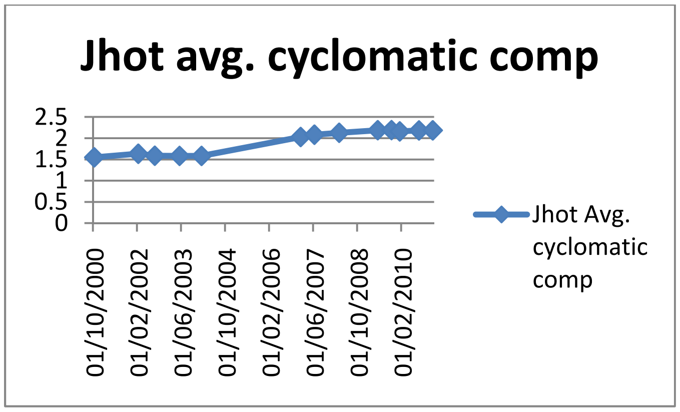

Figure 21. Graph showing average cyclomatic complexity for different releases of JhotDraw.

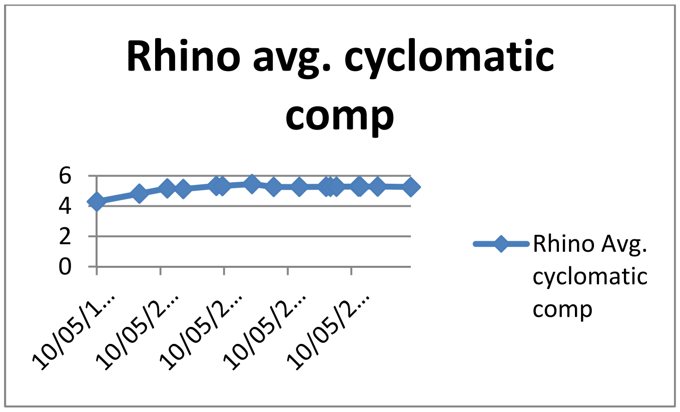

Figure 22. Graph showing average cyclomatic complexity for different releases of Rhino.

5.8 Law 3 Self Regulation

According to this law, the software evolution process is self regulating, leading to a steady trend. The observations made from the growth curve of JhotDraw and Rhino given in figure.1 and figure.2 shows a steady increase in size. There are no evidences of huge change or sharp increase in the size of the code.

6. RELATED WORK

Some of the prior work that closely relate to that of ours include the case study of Linux kernel by Ayelet Israeli el al[14]. Through the evolution study of 810 version of Linux kernel, their work highlighted the relatedness of Laws of software evolution to Linux kernel. A different view of evolution of linux was presented in [17][18] by Godfrey et al.

The pioneer work in the field of software evolution has been done by Belady and Lehman[1]. They conducted the study of 20 releases of OS/360 operating system. The study led them postulate the laws of software evolution. The laws were further developed and published in [3][5]. The work presented in this paper is also based on the laws of software evolution.

Stephen Cook et al [11] have given an elaborate explanation on classification of software in S, P and E type. In [19] Kemerer and Slaughter have presented the methodology to do empirical research in software evolution.

7. CONCLUSIONS AND FUTURE WORK

In this paper we presented the study based on two open source software. The applicability of Lehman's laws of software evolution to open source software was studied in the light of number of metrics. We observed that the reflection of some of the laws namely law 1, law2, and law 6 was easily determined using the metrics. But the relatedness of law 3, law 4 and law 5 to open source software system was hard to determine and will require more empirical studies with relevant data. The major contributions of this work are:

1. The study presented in the first of its type being done on open source software.
2. The validity of the study can be ascertained as the source is available on internet.
3. The study has opened up more opportunities for research in the field of software evolution.

8. REFERENCES

[1] Belady, L. A. and Lehman, M.M. 1976. A model of large program development. *IBM Syst. J.* 15, 225–252.

[2] Chidamber, S.R. and Kemerer, C. F. 1994. A metrics suite for object oriented design. *IEEE Transaction on. Software Engineering*. 20, 6, 476–493.

[3] Lehman, M.M. 1980. Programs, life cycles, and laws of software evolution. In *Proceedings of the IEEE (Special issue of Software Engineering.,* 68,9, 1060 – 1076.

[4] Lehman, M.M. 1980. On understanding laws, evolution, and conservation in the large-program life cycle. *Journal of Systems and Software.*1, 213-221.

[5] Lehman, M.M. 1996. Laws of software evolution revisited. In *Software Process Technology, ser. Lecture Notes in Computer Science.* 1149, 108-124. http://dx.doi.org/10.1007/BFb0017737

[6] McCabe, T.J. 1976. A complexity measure. *IEEE Trans. Software Eng.* 2, 4, 308–320.

[7] Home page jhotdraw7. [Online]. Available: http://www.randelshofer.ch/oop/jhotdraw/index.html.

[8] "Home page rhino." [Online]. Available: http://www.mozilla.org/rhino/.

[9] Cook, S., Harrison, R., Lehman, M.M. and Wernick, P. 2006. Evolution in software systems: foundations of the spe classification scheme. *Journal of Software Maintenance and Evolution: Research and Practice.*18, 1, 1-35.

[10] Jhotdraw project page. [Online]. Available:http://sourceforge.net/projects/jhotdraw/.

[11] Rhino revision details. [Online]. Available:https://developer.mozilla.org/en/Mozilla_ Source_ Code_ Via_ CVS.

[12] Israeli, A. and Feitelson, D.G. 2010. The linux kernel as a case study in software evolution. *Journal of Systems and Software*. 83, 3, 485 – 501.

[13] Basili, V.R., Briand, L. and Melo, W.L. 1995. A validation of object-oriented design metrics as quality indicators. *IEEE Transaction on Software Engineering*. 22, 751–761.

[14] Gyimothy, T., Ferenc, R. and Siket, I. 2005. Empirical validation of object oriented metrics on open source software for fault prediction. *IEEE Transaction on Software Engineering*. 31, 897–910.

[15] Godfrey, M.W. and Tu, Q. 2000. Evolution in open source software: A case study. In *Proceedings of the International Conference on Software Maintenance*. 131–142.

[16] Godfrey, M.W. and Tu, Q. 2001. Growth, evolution, and structural change in open source software. In *Proceedings of the 4th International Workshop on Principles of Software Evolution, ser. IWPSE '01. New York, NY, USA*. 103–106.

[17] Kemerer, C. and Slaughter, S. 1999. An empirical approach to studying software evolution. *IEEE Transactions on Software Engineering*. 25, 4, 493 –509.

.