Evolution of Cyclomatic Complexity in Object Oriented Software

Rajesh Vasa and Jean-Guy Schneider

Abstract— It is a generally accepted fact that software systems are constructed and gradually refined over a period of time. During this time, code is written and modified until stable releases of the system emerge. Many researchers have studied systems over a longer period of time in order to understand how they change and evolve. Despite these efforts, we still lack a precise understanding how various properties of software change over time, in particular in the area of object-oriented systems. Such an understanding is of great importance if we want to come up with techniques to provide feedback on the evolution of quality and predictions about further evolution of software systems. Historically, collection of sufficient data to build useful models was not practical as source code and build histories were not freely available. It is our opinion that by focusing our attention towards Open source software OSS) repositories, we have a better hope of building predictive models to help developers and managers. In this paper, we will report on an exploratory study analyzing object oriented OSS projects and present our findings based on this analysis.

Index Terms— Quantitative OO and design heuristics, OOD and quality characteristics assessment, Quantitative tracking of OO development activities.

1 Introduction

Software development is a highly iterative, dynamic process that requires constant feedback to gauge progress and make necessary adjustments to reach the required goal efficiently. Often software is built by many authors and it needs to be robust, flexible and must support the intrinsic need for distributed applications. These systems will need to be adaptive to new contexts and must evolve naturally throughout their lifespan.

Evolution is inherent in the nature of any real-world software system [5]. However, a survey of empirical research in software maintenance has found that less than two percent (2%) of empirical studies in software engineering focused on maintenance and much less on how software evolves over a period of time [4]. Research and studies into how software evolves is of great importance as it may assist us in building predictive models that can warn developers of impending danger or highlight decisions that may be unwise.

Past and recent studies into how software evolves has been mostly focused on large non-object oriented software systems such as Linux and/or Apache [4][5][7][8]. Over the last few years, some researchers in this field have started to focus their attention on how object oriented software systems evolve [9][10][11]. Despite the availability of a good amount of high quality software, very few researchers [1][7][13] have investigated Open source software (OSS). OSS is characterized by certain legal and pragmatic arrangements that ensure that the source code is generally available at no cost [1]. Another unique aspect in OSS projects is the development methodology used; active members make contributions without any direct financial incentive, often they are not in the same geographical location, and all communication is achieved using a combination of email, discussion boards and chat rooms. Studying Open source software provides a very unique perspective on software development as in this model most contributions to the project are voluntary. One can rationally argue that it must be well engineered to attract and maintain a sufficient number of contributors. Effectively, for a project to achieve fast growth and sustain the interest of active developers, it must be well structured and must exhibit a number of positive quality attributes.

To ensure that any software system does not deteriorate as it is evolves, we feel it is necessary to provide some feedback to the development team on a regular basis about how the software system is changing and evolving. We started our research with the aim of developing a predictive model that can monitor the code repository and provide general warnings to the development team. In this work, we present our findings and an initial hypothesis for a predictive model that has been built based on our observations of method evolution in object oriented code. The data for our study was collected from five different software systems, all developed using the Java programming language [16]. A short summary of the software systems we used in this case study is provided in Table 1 (refer to Table 6 for website URLs).

This paper is organized as follows: in Section 2, we discuss related work in the area of predictive models. In Section 3, we present the data that was collected for our study. Section 4 presents our observations and outlines the analysis that we performed on the data collected. In Section 5 we present our initial hypothesis. We conclude this paper in Section 6 with a summary of the main observations and outline future research directions.

- *Rajesh Vasa is with the Swinburne University of Technology, Hawthorn, VIC, Australia. E-mail: rvasa@swin.edu.au.*
- *Jean-Guy Schneider is with the Swinburne University of Technology, Hawthorn, VIC, Australia. E-mail: jschneider@swin.edu.au.*

2 Related Work

Software evolution as a research area started off with early

TABLE 1
SOFTWARE DESCRIPTION AND THE NUMBER OF VERSIONS ANALYSED

Name of Software	Description	Versions Analysed	Total number of methods	Observation period
JDictionary	English dictionary	7	568	6 months
JEdit	Text editor	10	4211	7 months
JasperReports	Report generation tool	8	1633	7 months
Tomcat	Web server and JSP container	9	5032	23 months
Hibernate	Object-relational persistence service	10	2629	2 months

work by Lehman [2]. Over the last 27 years he worked with many other researchers to propose and refine eight laws of software evolution [6][8] all of which are directly related to evolution of E-Type systems; that is, software systems that solve a problem or implement an application in the real-world, as in business systems. Kemerer and Slaughter [5] reviewed existing literature on software evolution and have designed an approach for longitudinal research that enlarges the scope of empirical data available on software evolution. They claim that "[software] evolution patterns could be examined across multiple levels of analysis (system and module), over longer periods of time, and could be linked to a number of organisational and software engineering factors" [5]. Godfrey and Tu [7] undertook a case study on evolution in OSS systems, focusing mainly on the Linux operating system. In their research they analysed the exponential growth of the Linux operating system and have concluded that it was made possible by the loose coupling and the component architecture supported by the operating system. Mockus et al. [1] studied the evolution of Mozilla and Apache, both OSS systems. They conclude that OSS projects benefit greatly from having a large team of testers, early-adopters and developers that can help isolate and replicate a defect quickly. They also argue that the design structure of those systems have a direct impact on the development speed; a highly modular, component based architecture allows fast evolution whereas a highly inter-dependent architecture generally requires a longer period of time between released versions.

3 DATA COLLECTION

Our research data was collected using JavaNCSS [14] a flexible source measurement tool for the Java language to collect raw metrics. In particular, we extracted method Cyclomatic complexity as defined originally by McCabe [12] as well as the non-comment source lines of code(NC-SLOC). JavaNCSS calculates Cyclomatic complexity by assigning every method a value of 1 initially, it then increments by one when the following Java keywords / statements are encountered: *if, for, while, case, catch.* It also increments by one when methods return abortively using the following keywords: *return* and *throw*. An ordinary return at the end of the method will however not be counted. These metrics were calculated and stored across all of the available versions for a particular software system. Our definition of a "version" is a stable software build that has been released by the development team, including both the binaries as well as the source code. This approach of assigning version numbers is similar to the strategy used by other researchers that have studied evolutionary trends [2]. We have assigned version numbers starting from 1 based on the release date. We have used this approach as our analysis view point would be similar to that of the development team that would assign the version numbers. Further, versions often signify that a set of milestones has been achieved and some of the defects found in earlier versions have been corrected. Although it is common for the development team to make use of major as well as minor version numbers we choose not to make that differentiation for the purpose of this initial case study. All software used in our case study was obtained directly from the development websites of the related project.

4 OBSERVATIONS AND ANALYSIS

We started our observations of the software systems by analysing the Non-Comment Source Lines of Code (NC-SLOC) as well as the total number of methods in each version. This data is shown in Table 2. The number of methods is shown in italics and enclosed in brackets next to the NC-SLOC measure. Our original goal was to study 10 released versions of all software systems, however due to time and resource limitations we were not able to obtain equal number of versions for all software systems. This initial data set was used to determine if there was some change in the software system being analysed. To ensure that changes in data do not cancel out each other, which can happen if equal number of lines are added and removed, we used the total number of method count in conjunction with NC-SLOC to ensure that there were some changes between versions.

In most cases we found the change in NC-SLOC to be between 7% and 100% when we compare the first version to the last version analysed in our data set for each system. The range of change was similar to this when the total number methods was counted in each version. An interesting observation was that the number of methods changed much more drastically over the period of evolution in all of the systems compared to NC-SLOC. This supports observations made by other researchers [15] where similar distribution patterns were noticed. Although the size fluctuated and dropped some times, over a longer period all software systems have exhibited some growth in volume. This observation can be seen to support Lehman's first law of software evolution [6][8] which states that "an E-type program that is used must be continually adapted else it becomes progressively less satisfactory." This observation also sup-

TABLE 2
LINES OF CODE AS MEASURED BY JAVANCSS

Versions	Non-comment Source Lines of Code (and Method Count)				
	Hibernate	JDictionary	JEdit	Jasper Reports	Tomcat
1	12915 (*2142*)	3810 (*346*)	43226 (*3917*)	5719 (*780*)	23898 (*3439*)
2	13807 (*2236*)	4034 (*368*)	43856 (*3968*)	5808 (*797*)	29170 (*4719*)
3	13954 (*2259*)	4464 (*444*)	44400 (*4031*)	5993 (*845*)	25468 (*3927*)
4	14120 (*2290*)	5070 (*508*)	44502 (*4034*)	7970 (*1245*)	25572 (*3820*)
5	14595 (*2318*)	5070 (*509*)	44911 (*4080*)	9156 (*1393*)	25586 (*3821*)
6	15871 (*2549*)	5206 (*519*)	45245 (*4112*)	10991 (*1585*)	26467 (*3906*)
7	16153 (*2595*)	6049 (*568*)	45350 (*4122*)	11159 (*1608*)	26504 (*3920*)
8	16180 (*2608*)		45355 (*4122*)	11712 (*1633*)	35037 (*4891*)
9	16255 (*2631*)		46164 (*4213*)		36588 (*5032*)
10	16330 (*2629*)		46272 (*4211*)		

TABLE 3
CYCLOMATIC COMPLEXITY DISTRIBUTION FOR TOMCAT OVER 9 VERSIONS

Cyclomatic Complexity	Percentage of methods (%) in each version									Average	St. Dev.
	1	2	3	4	5	6	7	8	9		
1	71.6	73.3	72.1	71.3	71.2	71.0	70.9	68.2	68.4	70.9	1.7
2,3,4	17.0	17.3	17.7	17.7	17.7	17.6	17.6	19.0	19.1	17.8	0.7
5,6,7	6.3	5.2	5.5	5.8	5.8	6.0	6.1	6.4	6.6	6.0	0.4
8,9,10	2.4	2.1	2.3	2.3	2.3	2.4	2.4	3.1	2.9	2.5	0.3
>10	2.6	2.2	2.4	2.9	2.9	3.0	3.0	3.3	3.1	2.8	0.4

ports similar data from Godfrey et al. [7] on their work measuring evolution of Open source software systems, in particular the Linux operating system.

After the initial observations based on volumetric measures, we moved to collecting Cyclomatic complexity for each non-blank method. To enable analysis of the data we broke the range of cyclomatic complexity information into five different categories. The data by category is shown in Tables 3 and 4, respectively. Cyclomatic complexity of one (1) was allocated a category of its own as in almost all projects that we analysed a large proportion of the methods fall into this category. The range was generally between 50% - 80% of the methods in any given project. In our study we counted the number of methods for each value of Cyclomatic complexity and then converted that into a percentage (%) value. All Cyclomatic complexity data in this report has been converted using this approach to allow us to compare the different projects in the study. This was necessary as the range of methods was quite vast between the various projects, smaller systems only have around 400 methods while larger projects contained around 5000 methods. The percentage values by category are shown in Tables 3 and 4. We have also presented in these tables the average for each category as well as the standard deviation. Only data from two projects have been shown in this report, the rest of the projects show a similar distribution pattern. As can be seen from the data here, the most interesting fact was the small value for the standard deviation in all categories. This data supports Lehman's Fifth law of software evolution [8], "Conservation of Familiarity", where Lehman argues that the content of successive versions is statistically invariant. Although the law was based on other measures, most of which were volumetric, it is interesting to note that complexity measures also support this particular statement. However, additional statistical tests are needed to further validate this fact.

5 HYPOTHESIS

Based on the observations collected in our experiment, we identified a recurring, quantitatively measurable evolutionary pattern that holds for all of the software systems that we studied. Our hypothesis can be summarised as follows:

1) In the worst case, the absolute difference between Cyclomatic complexities of successive versions seems to change around 4% for any given category.
2) The number of methods with a Cyclomatic complexity of 1 will be the highest as a percentage value for any given version. This value will be greater than 50%.
3) In absolute terms, the percentage of methods with a Cyclomatic complexity of 1 will generally change no more than 2.5% between successive versions. We have however noticed in our data set that in some instances the value will go over this range, but the chance of that happening was no more than 20% over the period of time we observed the software systems.
4) In absolute terms, the percentage of methods with a Cyclomatic complexity between 2 and 4 (inclusive) will generally change no more than 1.5% between successive versions.
5) In absolute terms, the percentage of methods with a Cyclomatic complexity between 5 and 7 (inclusive) will generally change no more than 0.75% between successive versions.

TABLE 4
CYCLOMATIC COMPLEXITY DISTRIBUTION FOR JASPER REPORTS OVER 8 VERSIONS

Cyclomatic Complexity	Percentage of methods (%) in each version								Average	St. Dev.
	1	2	3	4	5	6	7	8		
1	75.0	75.3	75.6	79.7	79.4	79.7	79.5	79.0	77.9	2.2
2,3,4	15.5	15.4	15.4	13.9	13.8	12.8	12.8	12.7	14.0	1.2
5,6,7	5.6	4.9	4.9	3.3	3.3	3.7	3.9	3.9	4.2	0.8
8,9,10	2.2	2.8	2.5	1.8	1.9	1.8	1.9	2.1	2.1	0.4
>10	1.7	1.8	1.9	1.5	1.9	2.2	2.2	2.6	2.0	0.3

6) In absolute terms, the percentage of methods with a Cyclomatic complexity between 8 and 10 (inclusive) will generally change no more than 0.5% between successive versions.
7) In absolute terms, the percentage of methods with a Cyclomatic complexity over 10 will generally change no more than 0.4% between successive versions.
8) The range of variation as described in points 2 – 7 will hold 90% of the time over the observed duration of the software system.

The above stated hypotheses were all built based on the data we have collected and by computing a set of ranges that best fit the data we collected. We have not used any formal mathematical techniques to construct our hypothesis, as this was only an initial hypothesis based on our observations of the data. The ranges that we have mentioned above fit our data set, but we observed that there is a 10% possibility that these ranges will not hold. This is the reason why we added the last hypothesis. However, we would like to contend that when the absolute difference in Cyclomatic complexity between two successive versions for any given category exceeds the range identified in our hypothesis, there must have been a substantial change between the two versions compared. This information can be used to detect a violation of Lehman's fifth law of software evolution. If the variation is greater than the range that we observed, it can be assumed that a disciplined development team will correct this anomaly within the next one or two versions. Not doing so would start to create a software system that is growing faster than the development team can keep up with. This also keeps in line with the expectation identified by Lehman et. al. in their eight law of software evolution [8].

In Figure 1, we have charted the Cyclomatic complexity distribution for one software system to support our hypothesis. The chart shows the range of cyclomatic complexity between 2 and 15. We have removed 1 as it is very large and would have made the graph difficult to understand. One of the key features of this graph is the negligible variation (between 1% and 2%) shown between successive versions of the same system. The graphs for all of the other systems are nearly identical and would have made the graph difficult to understand.

6 CONCLUSION

We started our research with the aim of developing an initial predictive model that can provide a warning to the development team when the software system starts to evolve abnormally. In this report, we have presented our observations of how methods evolve in object oriented systems, in particular focusing on the distribution of Cyclomatic complexity. We have put forward an initial set of hypotheses that indicate the normal variation between two successive versions of a software system, supporting Lehman's first and fifth law of software evolution. The first law expects to see a continuing change over time, the fifth law of software evolution states that there is a conservation of familiarity, i.e. the content of successive releases is statistically invariant. Though this was not an initial aim of our research, we hope that our initial results will facilitate automating the task of detecting violations of some of the known laws of software evolution. An interesting finding of our case study was that some of Lehman's laws are applicable to current generation object oriented software systems that were developed using OSS development methodologies. This was not expected as most of the laws were built based on observations of how large mainframe era software systems evolved. These software systems were built using a different set of methodologies and used non-object oriented languages and development technologies.

Future work will concentrate on revising our hypotheses and define a statistically sound predictive model. In our current study we used only 5 projects, but we would like to validate our improved hypothesis against a larger data set. Furthermore, it would be interesting to find out whether our work could be extended to see how classes, packages, components, and layers evolve within software systems. Once sufficient data is collected at various levels, we will be able to verify if Lehman's laws of software evolution hold for systems developed using object oriented programming languages other than Java. To improve the quality of our research effort as well as similar work undertaken by others, tool support will be necessary. In particular, the tools should be able to collect the required information directly from the code repositories used by the respective development teams. There is also much scope in this

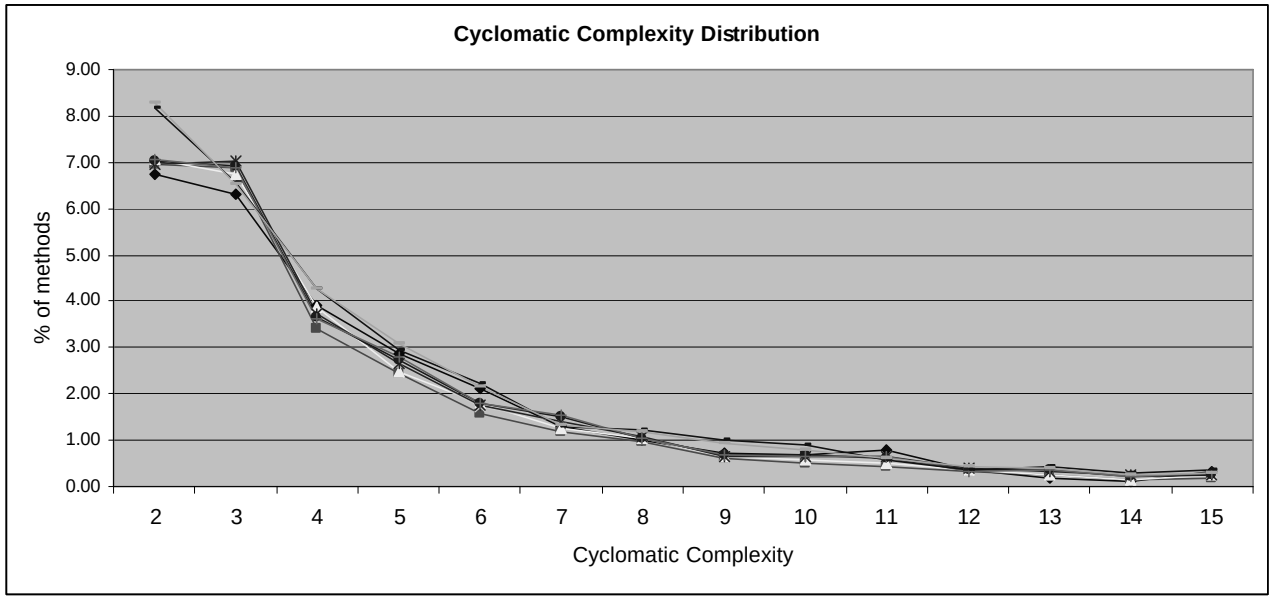

Figure 1. Cyclomatic complexity distribution for Tomcat (x-axis shows the complexity distribution between 2 and 15). Each line captures the distribution for a different version.

field to discover evolution patterns that can be used as the basis for other predictive models.

ACKNOWLEDGMENTS

The authors wish to thank Rajesh Kotha and Sai Panyam for their assistance in the data collection phase of this research project and Fernando Brito e Abreu for his detailed comments on an earlier draft.

REFERENCES

[1] A. Mockus, R. T. Fielding and J. D. Herbsleb. “Two Case Studies of Open Source Software Development: Apache and Mozilla.“ ACM Transactions on Software Engineering and Methodology, 11(3):1-38, July 2002.

[2] M. M. Lehman, “Programs, Life Cycles, and Laws of Software Evolution,” Proc. Special Issue Software Eng., IEEE, 68(9):1060-1076, 1980.

[3] N. F. Schneidewind, “The State of Software Maintenance,” IEEE Trans. Software Eng., 13(3): 303-310, Mar. 1987.

[4] C. F. Kemerer, “Empirical Research on Software Complexity and Software Maintenance,” Annals of Software Eng., 1(1):1-22, 1995.

[5] C. F. Kemerer and S. Slaughter, “An Empirical Approach to Studying Software Evolution,” IEEE Trans. Software Eng., 25(4):493-509, 1999

[6] L. A. Belady and M. M Lehman, “A Model of Large Program Development,” IBM Systems Journal, 15(1): 225-252, 1976.

[7] Michael W. Godfrey and Qiang Tu, “Evolution in Open Source Software: A Case Study,” Proceedings of the 2000 International Conference on Software Maintenance, San Jose, California, October 2000.

[8] M. M. Lehman, J. F. Ramil, P. D. Wernick, D. E. Perry and W. M. Turski. “Metrics and laws of software evolution – the nineties view.” In Proceedings of the Fourth International Software Metrics Symposium (Metrics ‘97), 1997.

[9] S. Demeyer, S. Ducasse and M. Lanza. “A hybrid reverse engineering approach combining metrics and program visualization." In Proceedings of the 6th Working Conference on Reverse Engineering (WCRE'99), pp. 175--186, 1999.

[10] T. Tamai T. and Nakatani. “An Empirical Study of Object Evolution Processes,” Proceedings International Workshop on Principles of Software Evolution (IWPSE'98), Kyoto, April 1998, pp. 33-37

[11] T. Tamai T. and Nakatani. “Analysis of Software Evolution Processes Using Statistical Distribution Models,” International Workshop on Principles of Software Evolution (IWPSE'02), Orlando, Florida, ACM, 2002, pp.120-123

[12] T. J. McCabe. “A complexity measure.” IEEE Trans. On Software Eng., SE-2(4):308-320, December 1976.

[13] G. Succi, J. Paulson & A. Eberlein. “Preliminary Results from an Empirical Study on the Growth of Open Source and Commercial Software Products.” Proceedings of the Third International Workshop on Economics-Driven Software Engineering Research (EDSER3), International Conference on Software Engineering 2001, Toronto, Ontario, Canada, May 12 - 19, 2001.

[14] http://www.kclee.com/clemens/java/javancss/

[15] H. Gall, M. Jazayeri, R. R. Klösch and G. Trausmuth. “Software Evolution Observations Based on Product Release History.” Proceddings of the International Conference on Software Maintenance (ICSM) 1997.

[16] J. Gosling, B. Joy, G. Steele and G. Bracha, "The Java Language Specification." Addison-Wesley, second edition, 2000.

TABLE 5
URL'S TO DOWNLOAD THE SOFTWARE WE USED IN OUR ANALYSIS

Product name	URL
Hibernate	http://hibernate.bluemars.net/
JDictionary	http://jdictionary.sourceforge.net/
Jasper Reports	http://jasperreports.sourceforge.net/
Tomcat	http://jakarta.apache.org/tomcat/
JEdit	http://jedit.sourceforge.net/