Observations on Patterns of Development in Open Source Software Projects[1]

Katherine J. Stewart
R. H. Smith School of Business
University of Maryland
College Park, MD 20742 USA
301-405-0576
kstewart@rhsmith.umd.edu

David P. Darcy
R. H. Smith School of Business
University of Maryland
College Park, MD 20742 USA
301-405-4900
ddarcy@rhsmith.umd.edu

Sherae L. Daniel
R. H. Smith School of Business
University of Maryland
College Park, MD 20742 USA
sdaniel@rhsmith.umd.edu

ABSTRACT
This paper discusses a project aimed at understanding how open source software evolves by examining patterns of development and changes in releases over time. The methodological approach of the research and initial observations are described. These include descriptions of release cycles and categorization of projects based on the overall changes in size and complexity exhibited across releases. Implications of these observations are discussed in light of prior and future work on understanding OSS evolution.

Categories and Subject Descriptors
D.2.8 [Software Engineering]: Metrics – *complexity measures*
D.1.5 [Programming Techniques]: Object oriented programming

General Terms
Measurement, Design, Languages

Keywords
Open Source Software, Software Evolution

1. INTRODUCTION
There are several qualitative and case study-based descriptions of how open source software is developed and evolves over time (e.g., [5, 6]). These often emphasize fast growth, fast release cycles and rapid improvement in quality (e.g.,[18]). One aspect of quality is software complexity.

Prior research has identified software complexity as a crucial factor in many important outcomes of the software development process including defect rates, maintainability, security, and reliability [11]. As these outcomes are often viewed as being at the root of what makes for "better" or "worse" software, complexity has been seen as a key contributor to overall software quality [17]. Understanding software complexity and how it may be managed throughout the software lifecycle is therefore of great interest to developers and researchers.

While most work to-date has examined software complexity in the context of closed source software development (e.g., [11, 12]), managing complexity could be at least as important in the open source development context because of the potentially higher fluidity in membership of the development team. In the open source context developers often are not bound to projects by employment relationships and therefore may come and go more often. Further, the impact on development that may occur from a lack of or lag in formal design specifications and documentation [19] may be moderated by complexity as less complex software will generally be easier to maintain. Thus minimizing complexity in OSS projects may have several benefits including facilitating new developers learning enough to contribute to the source code.

The goal of this study is to explore how software complexity develops in a large set of Open Source Software projects that vary in size and purpose. In doing this a secondary goal of the project is to provide some empirical data describing the overall patterns of development across these projects with regard to issues such as release frequency and rate of growth. The specific goal of this paper is descriptive. We attempt to address the question: what patterns of development and evolution in software complexity are found in OSS projects? By doing this we hope to provide a basis for more in-depth study into questions such as how the pattern of evolution in a project impacts other project outcomes and what factors influence which evolutionary pattern a project follows.

The next section provides a review of prior research on software complexity followed by a discussion of the special case of open source software. The literature review is followed by a description of the data collection and analytical methods employed in the study. The study collected and analyzed data on over 1,000 releases of more than 200 open source projects,

Open Source Application Spaces: Fifth Workshop on Open Source Software Engineering (5-WOSSE) May 17, 2005, St Louis, MO, USA.

focusing on three key measures of complexity: size, coupling, and cohesion. Initial findings regarding the patterns displayed by these projects and preliminary explorations of the possible relationships among our observations are described. In the final section of the paper we discuss implications and directions for future research.

2. PREVIOUS RESEARCH

2.1 The Structural Complexity of Software

The structural complexity of software has been defined as "the organization of program elements within a program" ([9], p. 191). The term element can refer to a procedure, function, method, module, class and so on. The definition implies a program level of analysis, which sets it apart from conventional complexity analyses that tend to focus on a single element (such as a single module or a single class). In theory, software can be structured in an infinite number of ways to solve the same underlying problem.

Solving a problem through software development proceeds through modularization, that is by breaking down the problem into constituent parts and implementing those parts. As design, implementation and maintenance decisions are made, the particular choice of part delineation and content has corresponding impacts on the structural complexity of the software. More formally (see [4]), the structural complexity of a program is a function of the composition of individual parts (e.g., cohesion) and on the associations among parts (e.g., coupling).

2.2 Software Evolution

Software evolution occurs when an external stimuli leads to a change in the software. It was originally conceptualized as capturing change activity after development of the software was complete (i.e. during the maintenance phase of the software lifecycle [1, 13]). More recently, the notion of evolution has been applied across the entire software life cycle [2]. The Laws of Software Evolution [1] represent a seminal work in this area. Of direct relevance to this research is the second law which states that as systems evolve, complexity increases unless it is actively managed ([15], p. 18). The management of complexity, referring to perfective or anti regressive rework, is an increasingly neglected step under tighter project time constraints in closed source contexts. Nevertheless, given the increasingly complex nature of problems being tackled with software, controlling complexity must be attended to as software continues to grow in functionality and size [7].

2.3 Complexity and Evolution in OSS Development

Much of what we know about software comes from analyses of closed source development. For example, the COCOMO models were developed in a closed source context [3]. Some of these models have been applied to open source with appropriate cautions. For example, if closed source conditions could be applied to the development of Debian 2.2, it would have cost almost $1.9 billion with a schedule of more than 6 years (see [8]). It is uncertain if what we know about closed source software can be applied to open source software [20]. One comparative analysis of three open source and three closed source projects [16] found that both types of projects grew at about the same rate and that the structure in terms of modularity was no different; however, creativity was more prevalent in the open source projects and defects were found and fixed more rapidly in the open source projects.

Past work evaluating OSS code has provided analyses of some of the largest, most widely used OSS projects such as Linux and Apache. This work has found evidence that OSS does not necessarily follow the laws of software evolution developed by Lehman. For example, [5, 6] found that Linux continued to grow at a geometric rate even after attaining a very large size. They suggest that better understanding of this pattern is obtained from examining subsystems separately. For example, the drivers subsystem exhibited greatest growth, and this may be possible because of the nature of drivers (being relatively self contained); the kernal itself is not nearly so big nor has it experienced as much growth as might be presumed from an analysis of the entire system. Similarly, [14] found that while the GNOME project overall displayed continuing growth, it seemed to be accounted for by newer subprojects with some components displaying the expected pattern of tapering growth.

One implication of these findings is that understanding patterns of development and change in complexity may benefit from analysis at the level of relatively clearly delineated projects or subprojects. In other words, to understand how the complexity of OSS code changes over time, it is important to focus on development of a single code base in addition to considering how a large project grows by adding isolated (from the perspective of dependency among the code) subprojects.

3. METHODS AND RESULTS

Based on the discussion above, we chose to study size, coupling, and cohesion as indicators of complexity that are relevant regardless of the development or distribution process of the software (i.e., open or closed), and we focused on projects that could be identified as developing a single code base, rather than "umbrella" type projects such as GNOME. The empirical study proceeded in several stages: sample selection, data collection, measure calculations, an iterative data cleaning process, categorization of projects into basic evolutionary patterns, and exploratory analysis of relationships among the pattern categorization and other variables.

3.1 Sample Selection

The sample was drawn from projects that provide code using SourceForge.net (www.sourceforge.net). There may be differences across programming language that would impact the development of complexity in software. To limit these differences but still allow some assessment of their potential impact, we sampled projects that used one of two languages, either C++ or Java. The problem domain of the software may also introduce variance in complexity; we limited our sample to the domains with the largest number of projects on SourceForge.net, Internet and Networking. From this set we eliminated projects if they had not released at least one version of code because our analysis relies on examination of software code, thus if there is no code we cannot calculate any of the

measures of interest. Projects were also eliminated if the project description on SourceForge.net indicated that it included multiple separate development streams or subprojects. In order to focus on projects that represent the open source movement we only included projects that use OSI approved licenses. Applying these criteria, we gathered data on 216 projects.

3.2 Data Collection and Measurement

We developed automated scripts to download the source code for all releases available on SourceForge.net for each project in our sample. We then used Scientific Toolworks' Understand software to compute complexity measures for each version of each project. The main measures produced by this tool and used in our study were size in terms of lines of code (LOC) and measures of coupling and cohesion. Understand calculates the number of classes each class is coupled to and the percent cohesion of each class. We reverse code the cohesion measure by subtracting it from 100 so that increases in both coupling and cohesion (labeled lack of cohesion when reversed) represent increases in complexity. In order to create a release level measure for coupling we average all coupling measures across all classes in a given version of a given project. The same procedure is used for our release level measure of lack of cohesion.

3.3 Data Integrity

We took several steps to ensure that the releases downloaded from the internet for a single project on SourceForge.net represented a single development stream on one project. The data was cleansed using two main approaches. The first approach focused on the naming conventions for each release of a project. We examined the file name of each release to identify cases in which multiple naming conventions indicated releases that did not represent sequential development of a single code base. For example if a project had several releases with a common base name and several other releases with a different common base name we attempted to determine if these actually represented parallel development streams. Parallel development streams found in projects in the sample existed for different operating systems, different spoken languages, and related client and server software. For these cases we retained the development stream that had the greatest number of releases and excluded the other from our analyses.

The second approach focused on identifying files that may not belong in the set of releases for a project by examining the pattern of development in each project over time. If a project's LOC for sequential releases had a repeating pattern of increases and decreases that represented large changes from one version to the next, we then reviewed public forums, web pages and discussion groups associated with the project. The purpose of this review was to try to understand why the variance occurred and cleanse the data as appropriate. For example, in one case we discovered that, while the naming convention of the project appeared to the uninitiated to represent a single stream of development, odd and even release numbers actually represented parallel development streams, one a "stable" stream and one a "development" stream. In these cases we again retained the stream with the largest number of releases in order to maximize the sample size.

3.4 Behavior Patterns of a "Typical" Project

The average project in our sample released 7.5 versions of the software over a period of 354 days, with the first release posted to SourceForge.net approximately 68 days after the project had been registered there, and subsequent releases every 39 days. Most of these releases increased in size, however on average each project had one release that had no change in LOC and one release that had a decrease in LOC. Descriptive statistics are included in table 1. These show that there were some significant differences between the absolute levels of variables across C++ and Java projects. When we examined patterns of change rather than absolute levels of variables, there were no significant differences. Paired samples t-tests indicated that both C++ and Java projects had increases in LOC and coupling from the first to the last release ($p<.05$ in all cases), but changes in cohesion and the ratio of comments to code were not statistically significant for either sub-sample or the overall set.

3.5 Variability in Behavior Across Projects and Preliminary Categorizations

Our focus on the evolution of code quality, specifically as indicated by structural complexity, led us to categorize projects into 4 groups: those that displayed no change in coupling or cohesion (including 56 projects that had only one release), those projects where both coupling and cohesion measures indicated improved quality between the first and last releases, those projects where both coupling and cohesion measures indicated deteriorating quality between the first and last releases, and those projects where one type of complexity improved and the other deteriorated. There was no significant difference in the percentage of projects in each category based on programming language, thus table 2 shows the number of projects in each category across the entire sample.

Table 1: Descriptive Statistics

	Mean (N=216)	C++ (N=111)	Java (N=105)	p
Releases	7.5	8.3	6.6	.444
Days between releases	38.5	27.7	49.9	.007
First release LOC	7614.2	5323.7	10035.6	.086
Last release LOC	11084.1	6843.0	15567.7	.008
First release Coupling	2.6	2.7	2.5	.305
Last release Coupling	2.8	3.0	2.7	.287
First release Lack of cohesion	45.4	52.7	37.8	.000
Last release Lack of cohesion	45.0	51.9	37.7	.000
First release Comment lines	3432.8	2246.2	4687.2	.040
Last release Comment lines	5343.7	2858.7	7970.7	.002

Table 2: Categorization Based on Overall Changes in Coupling and Lack of Cohesion

Category	N	%
1. No change in Coupling and Lack of Cohesion	71	32.9
2. Coupling and Lack of Cohesion Decrease	23	10.6
3. Coupling and Lack of Cohesion Increase	56	25.9
4. Opposite changes in Coupling and Lack of Cohesion	66	30.6

The next step in our analysis was to try to understand what factors may be associated with the categorization of projects. We conducted some preliminary analyses assessing potential effects of initial complexity, release pattern (i.e., average time between releases, number of releases, size change across releases), programming language, and use of comments, but relationships to the project categorization were not statistically significant. Because of potential normative implications for managing code quality, factors that lead a project to be in category 2, where both measures of complexity decrease, may be especially interesting. Thus we conducted preliminary investigations using qualitative information on the projects in this set (e.g., by reading their release notes). While some projects in this set did seem to purposely manage complexity, there was not a clear pattern of such management obvious across all projects in the set.

6. CONCLUSIONS, LIMITATIONS, AND FUTURE DIRECTIONS

Given the intricate nature of software development, it is not surprising that the preliminary analyses mentioned above failed to reveal a simple connection between initial conditions and later project categorizations. As stated in the introduction, the main purpose of this paper was descriptive in nature. There are at least three interesting observations we believe can be taken from the analysis to-date. The main point of interest is that the data provides empirical evidence that a subset of OSS projects manage to improve on these quality measures at the same time as they are increasing in size (19 of the 23 projects in category 2 showed overall increases in LOC). Similarly, there is a larger set of projects in category 4 that improves one complexity measure. These observations provide some empirical support for the notion that the OSS development process may lead to on-going quality improvements and suggests specific measures by which such quality improvement may be analyzed. Similarly, the results demonstrate the unsurprising fact that not all OSS projects achieve such improvements, suggesting that using an open development process is by no means a guarantee of continuous improvement.

A second observation of interest arises from the process we followed to enhance data integrity. We discovered that projects used the SourceForge.net repository in significantly different ways. Some maintained clearly labeled and organized code files whereas others combined many different kinds of files without a clear organizational scheme. Our experience that every project had to be manually scrutinized highlights the need for caution in using the SourceForge.net data, as others have also noted [10]. However, it is possible that this variance itself may be an interesting source of data on projects. For example, the organization of the project files in the repository may be related to the evolution of the project by influencing the level of difficulty that new developers or users may encounter when joining a project. This is one area in which we intend to extend our investigation.

A third observation that may be of interest to others studying the evolution of OSS projects comes from the descriptive data in table 1. The table provides one possible baseline that might be used for assessing the frequency of releases and rates of change in basic descriptive measures of OSS code so that, for example, what projects use "faster" or "slower" release cycles may be quantified. Of course, the usefulness of this data for that purpose is subject to the limitations of our sampling procedure as described above. Similarly, the view of improvement and deterioration discussed in the paper is limited by the relatively simplistic categorization scheme employed.

Next steps in this project will focus on refining the categorization of projects using cluster analysis and functional data analysis and combining this data with other data (e.g., the number of developers working on the projects and indicators of activity on the projects) to consider a larger set of potentially relevant factors influencing their evolution. By following this path we may be able to build on the work of others, such as [14] who observed that increasing size in LOC was highly correlated with increasing size of the development group, and [21] who surveyed projects on SourceForge.net and Freshmeat.net and suggested that the change management and tracking tools provided by the websites were commonly used for quality management.

7. ACKNOWLEDGEMENTS

This paper is based on research supported by the National Science Foundation award IIS-0347376 and the University of Maryland Center for Electronic Markets and Enterprises. All opinions expressed are those of the authors. We thank Julie Inlow, Chang-Han Jong, and Vincent Kan for their invaluable research assistance on this project.

8. REFERENCES

1. Belady, L.A., and Lehman, M.M. A Model of Large Program Development. *IBM Systems Journal*, 3, (1976) 225-252.
2. Bennett, K.H.; Knight, C.; Munro, M.; and Xu, J. Centres of Excellence: Research Institute in Software Evolution, University of Durham. *Computing and Control Engineering Journal*, 11, 4 (2000) 179-186.
3. Boehm, B. *Software Engineering Economics*. Englewood Cliffs, NJ: Prentice Hall, 1981.
4. Darcy, D.P.; Kemerer, C.; Slaughter, S.A.; and Tomayko The Structural Complexity of Software: An Empirical Test. *University of Maryland, Smith School of Business Working Paper*, (2005)
5. Godfrey, M.W., and Tu, Q. Evolution in Open Source Software: A Case Study, in *Proceedings of International*

Conference Software Maintenance, San Jose, California, 2000, 131-142.
6. Godfrey, M.W., and Tu, Q. Growth, Evolution, and Structural Change in Open Source Software, in *Proceedings of International Conference on Software Engineering, 4th International workshop on Principles of Software Evolution*, 2002, ACM Press 103-106.
7. Goldin, D.S. Taming Software Complexity Is Critical, *Design News*, Vol. 56, No. 1, January 8 (2001), 172.
8. Gonzalez-Barahona, J.M.; Perez, M.A.O.; Quiros, P.d.l.H.; Gonzalez, J.C.; and Olivera, V.M. Counting Potatoes: The Size of Debian 2.2, *Upgrade Magazine*, Vol. II, No. 6, (2001),
9. Gorla, N., and Ramakrishnan, R. Effect of Software Structure Attributes Software Development Productivity. *Journal of Systems and Software*, 36, 2 (1997) 191-199.
10. Howison, J., and Crowston, K. The Perils and Pitfalls of Mining Sourceforge, in *Proceedings of International Conference on Software Engineering, Mining Software Repositories Workshop*, Edinburgh, 2004.
11. Kemerer, C.F. Software Complexity and Software Maintenance: A Survey of Empirical Research. *Annals of Software Engineering*, 1, 1 (1995) 1-22.
12. Kemerer, C.F., and Slaughter, S.A. An Empirical Approach to Studying Software Evolution. *IEEE Transactions on Software Engineering*, 25, 4 (1999) 493-509.
13. Kemerer, C.F., and Slaughter, S.A. An Empirical Approach to Studying Software Evolution. *TSE*, 25, 4 (1999) 493-509.
14. Koch, S., and Schneider, G. Results from Software Engineering Research into Open Source Development Projects Using Public Data. *Diskussion zum Tagigkeitsfeld Informationverarbeitung und Informationswirtschaft*, 22, (2000) 1-16.
15. Lehman, M.M., and Ramil, J.F. An Approach to a Theory of Software Evolution, in *Proceedings of Proc. 2001 Intern Workshop on Principles of Software Evolution*, 2001.
16. Paulson, J.; Succi, G.; and Eberlein, A. An Empirical Study of Open Source and Closed Source Software Products. *IEEE Transactions in Software Engineering*, 30, 4 (2004) 246-256.
17. Prahalad, C.K., and Krishnan, M.S. The New Meaning of Quality in the Information Age. *Harvard Business Review*, (1999) 109-118.
18. Raymond, E.S. *The Cathedral and the Bazaar: Musing on Linux and Open Source by an Accidental Revolutionary*. Sabastopol, CA: O'Reilly, 2001.
19. Scacchi, W. Understanding the Requirements for Developing Open Source Software Systems. *IEE Proceedings on Software*, 149, 1 (2002) 24-39.
20. Scacchi, W. *Understanding Open Source Software Evolution*, (2004). http://www.ics.uci.edu/~wscacchi/Papers/New/Understanding-OSS-Evolution.pdf
21. Zhao, L., and Elbaum, S. Quality Assurance under the Open Source Development Model. *Journal of Systems and Software*, 66, (2003) 65-75.