IMPROVING OPEN SOURCE SOFTWARE MAINTENANCE

VISHAL MIDHA
The University of Texas – Pan American
Edinburg, TX 78539

PRASHANT PALVIA
The University of North Carolina at Greensboro
Greensboro, NC 27402

RAHUL SINGH
The University of North Carolina at Greensboro
Greensboro, NC 27402

NIR KSHETRI
The University of North Carolina at Greensboro
Greensboro, NC 27402

ABSTRACT

Maintenance is inevitable for almost any software. Software maintenance is required to fix bugs, to add new features, to improve performance, and/or to adapt to a changed environment. In this article, we examine change in cognitive complexity and its impacts on maintenance in the context of open source software (OSS). Relationships of the change in cognitive complexity with the change in the number of reported bugs, time taken to fix the bugs, and contributions from new developers are examined and are all found to be statistically significant. In addition, several control variables, such as software size, age, development status, and programmer skills are included in the analyses. The results have strong implications for OSS project administrators; they must continually measure software complexity and be actively involved in managing it in order to have successful and sustainable OSS products.

Keywords: OSS, Complexity, Software Maintenance

INTRODUCTION

The importance of software maintenance in today's software industry cannot be underestimated. Maintenance is inevitable for almost any software. Software maintenance is required to fix bugs, to add new features, to improve performance, and/or to adapt to a changed environment. Pigoski [39] illustrated that the portion of industry's expenditures used for maintenance purposes was 40% in the early 1970s, 55% in the early 1980s, 75% in the late 1980s, and 90% in the early 1990s. Over 75% of software professionals perform program maintenance of some sort [24]. Given the numbers, the understanding of software maintenance is prudent.

It is not unusual that a developer modifying the source code has not participated in the development of the original program [31]. As a consequence, a large amount of the developer's efforts goes into understanding and comprehending the existing source code [46]. Comprehending existing source code, which involves identifying the logic in and between various segments of the source code and understanding their relationships, is essentially a mental pattern-recognition by the software developer and involves filtering and recognizing enormous amount of data [43]. As software is becoming increasingly complex, the task of comprehending existing software is becoming increasingly tough [43]. Fjelstad and Hamlen [17] reported that more than 50% of all software maintenance effort is devoted to comprehension. The comprehension of source code, thus, plays a prominent role in software development.

In this article, we examine software complexity and its impacts in the context of open source software (OSS). Past efforts have been piecemeal or based on limited information. For example, comprehension of the source code has been linked with source code complexity. The empirical evidence on the magnitude of the link is relatively weak [29]. However, many such attempts are based on experiments involving small pieces of code or analysis of software written by students [2]. In order to remedy this situation, we analyze real world software written by the OSS developer community. A number of studies has examined the impact of complexity on maintainability and made recommendations to reduce the complexity [30][31]. But, no study, to the best of our knowledge, has tested if the reduced complexity was actually beneficial to the developers performing software maintenance. This study specifically examines the impact of *change* in software complexity on maintenance efforts.

Open Source Software Development

A typical open source project starts when an individual (or group) feels a need for a new feature or entirely new software, and someone in that group, eventually writes one. In order to share it with others who have similar needs, the individual/ group releases the software under a license that allows the community to use, and to see and modify the source code to meet local needs and improve the product by fixing bugs. Making software available widely on an open network, e.g., the Internet, allows developers around the world to contribute code, add new features, improve the present code, report bugs, and submit fixes to the current version. The developers of the project incorporate the features and fixes into the main source code and a new version of the software is made available to the public. This process of code contribution and bug fixing is continued in an iterative manner as shown in Fig 1.

OSS supporters often claim that OSS has faster software evolution. The idea is that multiple contributors can be writing, testing, or debugging the product in parallel. Raymond [42] mentioned that more people looking at the code will result in more bugs found, which is likely to accelerate software improvement. The OSS model claims that the rapid evolution produces better

Received: June 22, 2009 Revised: August 17, 2009 Accepted: September 9, 2009

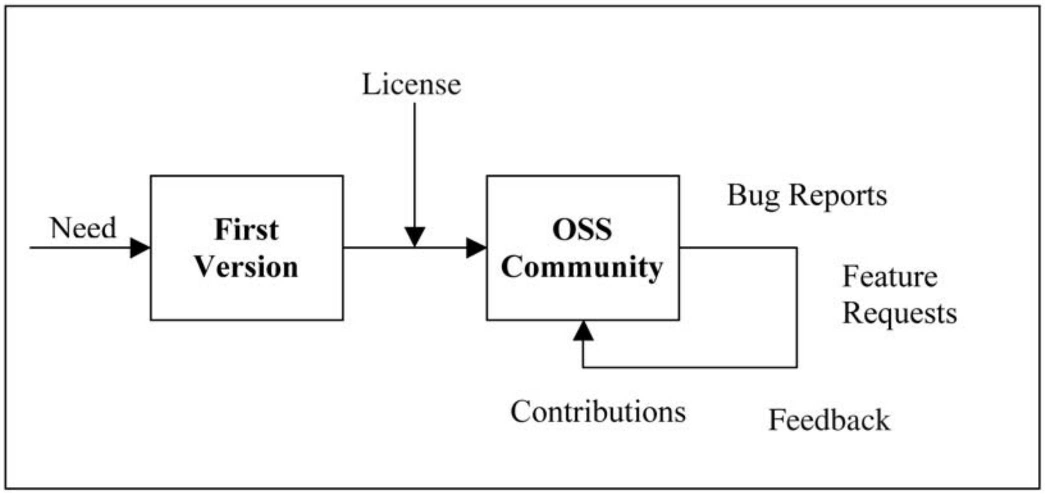

FIGURE 1 — OSS Development

software than the traditional closed model because in the latter "only a very few programmers can see the source and everybody else must blindly use an opaque block of bits" [38].

One interpretation of the OSS development process is that of a perpetual maintenance task. Developing an OSS system implies a series of frequent maintenance efforts for bugs reported by various users. As most of the OSS projects are results of voluntary work [14][48], it is crucial to ensure that such volunteers are able to work with minimal effort. The motivation for why developers contribute to a source code has received a great deal of attention from researchers [34]. However, the factors that can make the OSS community to not contribute to a source code have received limited attention.

In this light, von Hippel and von Krogh [51] noted that the major concern among developers was the complexity of the source code and the level of difficulty of the embedded algorithms. Fitzgerald [15] pointed that increasing complexity posits a barrier in the OSS development and may trigger the need for either substantial software reengineering or the entire system replacement. Therefore, it is vital to understand the complexity of the source code and its impact on software development, and even more importantly, on OSS development.

OSS and Complexity

A complex project, in general, demands a large share of resources to modify and correct. When the source code is easy, it is easier to maintain it. On the contrary, when a source code is complex, developers have to expend a large portion of their limited time and resources to become familiar with the source code. In OSS, where the developers seek to gain personal satisfaction and value from peer review and are not bound to projects by employment relationships, they have the option to leave the project at any time and join other projects where their resources could be used more efficiently. Therefore controlling complexity in OSS projects may have several benefits, including facilitation of new developers' learning. Feller and Fitzgerald [14] pointed that if new contributors are to have any chance at contributing to OSS projects, they should be able to do so with minimal effort. Controlled complexity helps achieve that; thus being indispensable for OSS [14].

Much of what we know about software complexity comes from analyses of closed source development (e.g., [5]). As noted by Stewart et al [49], even though the results from those findings have been applied to OSS (e.g., study of Debian 2.2 development [21]), there remains a relative scarcity of academic research on the subject. More importantly, these studies were limited to a small number of projects.

The remainder of the paper is organized as follows: The next section draws on relevant literature to develop a theoretical model. It is followed by a description of the methods and measures used in the study. The following sections present the evaluation of the model and discussion of the results. The paper is concluded by acknowledging its limitations and highlighting its contributions to both research and practice.

MODEL DEVELOPMENT

Basili and Hutchens [4] define complexity as a measure of the resources expended by a system while interacting with a piece of software to perform a given task. It is important to clearly understand the term "system" in this definition. If the interacting system is a computer, then complexity is defined by the execution time and storage required to perform the computation. For example, as the number of distinct control paths through a program increases, the complexity may increase. This kind of complexity is called "Computational Complexity" [11]. If the interacting system is a programmer, then complexity is defined by the difficulty of performing tasks. This complexity comes from "the organization of program elements within a program" [22], for example, tasks such as coding, debugging, testing, or modifying the software. This kind of complexity is known as "Cognitive Complexity". Cognitive complexity refers to the characteristics of the software which make it difficult to understand and work with [11]. It is our primary concern.

The notion of cognitive complexity is linked with the limitations of short term memory. According to the cognitive load theory, all information processed for comprehension must at some time occupy short-term memory [43]. Short term memory is described as the capacity of information that brain can hold in an active, highly available state. Short term memory can be thought of as a container, where a small finite number of concepts can be stored. If data are presented in such a way that too many concepts must be associated in order to make a correct decision, then the risk of error increases. In OSS, a voluntary developer must retain the existing source code in short term memory in order to successfully modify the existing code. The capacity of holding information may vary depending on the individual and may limit the capability of developers to comprehend and modify the existing source code. Kearney et al [29] suggested that the difficulty of understanding depends, in part, on structural properties of the source code. As we are concerned with the impact of complexity on source code comprehension, we focus on properties related to the source code. This argument forms the basis for theorizing the impact of complexity on various aspects on OSS development, as described below.

Number of Bugs

The main idea behind the relationship between complexity and number of bugs is that when comparing two different solutions to the same problem, all other things being equal, the more complex solution will generate more bugs. This relationship is one of the most analyzed by software metrics' researchers and previous studies and experiments have found this relationship to be statistically significant [11][27].

In order for a programmer to understand the existing source code, he needs to understand the flow of logic. And, when a programmer has to deal with a source code with high cognitive complexity, he has to *frequently* search among dispersed pieces of code to determine the flow of logic [40]. Understanding and recollecting such dispersed pieces increase the cognitive load on the programmer making complex code maintenance more liable to human errors. Complex software, hence, need more maintenance efforts. Gill and Kemerer [20] reported that the number of bugs in a program is positively associated with maintenance effort and recommended further empirical testing with a larger data set. Therefore OSS projects which experience increase in complexity over its previous version also would experience an increase in the number of bugs (over its previous version). Based on above, we propose:

H1: *An increase in the source code's cognitive complexity is positively associated with an increase in the number of bugs in the OSS source code.*

Contributions from New Developers

Because of the important role of volunteer developers in the OSS development, attracting new developers and keeping them motivated is crucial to OSS development. Keeping the developers motivated is especially important during the early development stage so that the number of developers can reach a critical mass. Some of the cited developers' motivations include intellectual gratification, career future incentives, learning and enjoyment, ego-boosting, and peer recognition [6][8][35][37].

Once a new developer is motivated to voluntarily contribute, he needs to first spend a large amount of time and resources to understand the existing source code. When the source code is easy to comprehend, it is easier to modify. However, when the source code is complex, a developer is required to invest additional effort and resources to understand it. Devoting such effort and resources may pose a barrier to the developer's motivation to contribute. Such a barrier may lead the potential developer to not contribute to the project at all, or, in worst case, to leave the project. Hence,

H2: *An increase in the source code's cognitive complexity is negatively associated with an increase in the number of contributions to the OSS source code from new developers.*

Time to Fix Bugs

More complex source code adds to a programmer's cognitive load [12]. High cognitive load requires more time-consuming and resource-demanding effort to familiarize oneself with the code. It is even possible that a source code is so complex that it cannot be comprehended at all. In such a scenario, the programmer may spend time and resources on other activities, thereby further lowering the productivity of the project.

In other words, a source code with lesser cognitive complexity does not need as much effort or resources, thus reducing the turnaround time required to fix repairs. This leads to the next hypothesis that OSS projects which experience increase in cognitive complexity over its previous version require longer time to fix the bugs. Hence, we hypothesize,

H3: *An increase in the source code's cognitive complexity is positively associated with an increase in the average time taken to fix the bugs in OSS source code.*

Combining all the preceding conceptual arguments gives the research model shown in Fig. 2. Note that several control variables have been included in the model in order to increase the robustness of our findings. The specific variables will be described in the next section.

METHODS

The following explanation is helpful in understanding the research design and methods. This study investigates the impact of *change* in complexity. To compute the change in complexity, the complexity of two consecutive versions of software must be looked at. It is important to note that the complexity of the source code of a software version can only be measured after it has been released to the OSS community. Only after it has been used, the discovered bugs are reported and the code is modified to fix these bugs. Once significant amount of modifications have been made, a new version is released to the public. Due to the modifications in the source code, the complexity of the source code changes. In order to compute the change in complexity of the current version (say N^{th}) from its previous version ($N\text{-}1^{th}$), one needs to measure the complexity of both the current (N^{th}) and the previous version ($N\text{-}1^{th}$). As the modifications and contributions made to the current version (N^{th}) are available in the next version ($N+1^{th}$), one needs to also look at the next version ($N+1^{th}$) to find these modifications and contributions. As a consequence, for each project, we need to study three releases, referred to as the first ($N\text{-}1^{th}$), the second (N^{th}), and the third ($N+1^{th}$).

OSS projects hosted at SourceForge were examined in this study. SourceForge is the primary hosting place for OSS projects which houses about 90% of all OSS projects. It has been argued SourceForge is the most representative of the OSS movement, in part because of its popularity and the large number of developers and projects registered [23][54]. Researchers interested in investigating issues related to the OSS phenomenon have predominantly used SourceForge data [23][51][54].

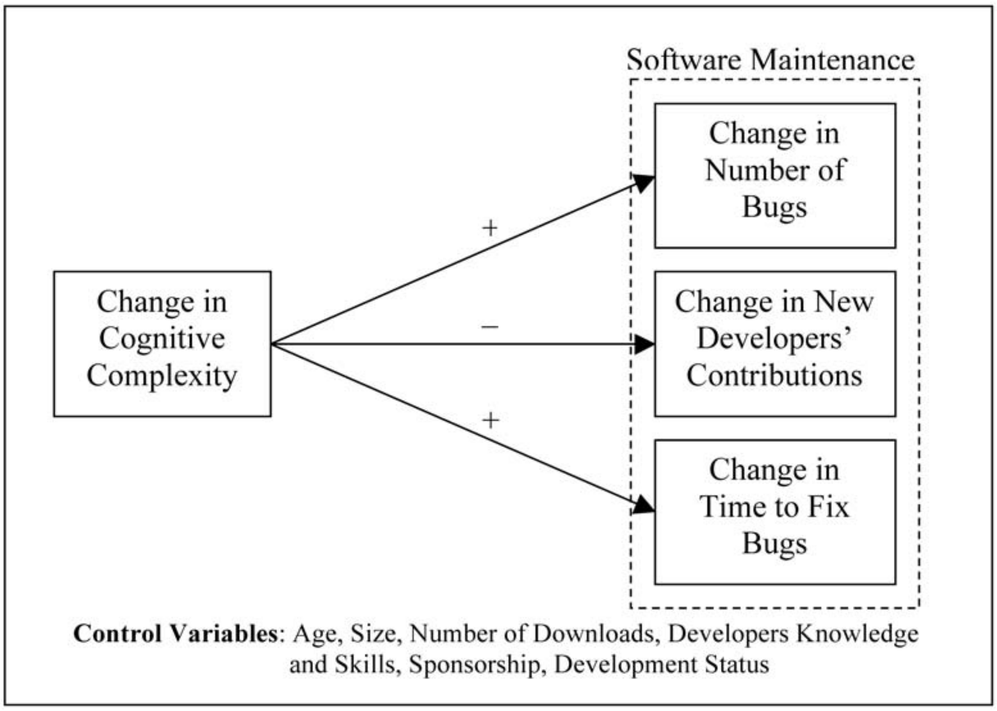
Software Maintenance
Change in Cognitive Complexity
+
−
+
Change in Number of Bugs
Change in New Developers' Contributions
Change in Time to Fix Bugs
Control Variables: Age, Size, Number of Downloads, Developers Knowledge and Skills, Sponsorship, Development Status

FIGURE 2 — The Research Model

Studying all projects hosted on SourceForge was unfeasible and impractical due to resource limitations. Data selection was limited to projects that were targeted to either end users or developers. In order to avoid ambiguity, projects that were targeted to both end users and developers were excluded. Further selection was made by controlling for the programming language and the operating system. Past literature suggests that programming language has an explicit impact on complexity [52] and program size [28]. It is also difficult to compare lines of code between "high" and "low" level programming languages. Lower level programming languages have more lines of code and take longer to develop than higher level programming languages. As C family of languages is the most preferred by the OSS developers [45], only projects written in C/C++ or multiple languages including C/C++ were selected. Secondly, operating system of the project impacts the complexity of the software and the development effort required. To encompass majority of the projects targeted for developers and end users, all projects in the data set were designed either for the Windows or the Linux/Unix operating system.

As the data was collected from three different versions of software, the sample was further restricted to the projects that had at least 3 versions. A version released between first 3-months of the registration date is considered First release, another major version released between 3 to 6 months of its registration date is considered Second release, and yet another major version released within 6 to 12 months of its registration date is considered the Third release for this study. Therefore, to be able to get the data for three different versions, we considered all projects that were registered between SourceForge between January 2003 and August 2006 so that the third release for the projects that were registered in August 2006 was released by August 2007. The final data collection was completed in August 2007. Lastly, projects were chosen for which the required data were publicly available (not all projects allow public access to the bug tracking system). Following the above criteria, the final sample size was limited to 450 projects.

MEASURES

Cognitive Complexity

McCabe's cyclomatic complexity (CC) assesses the difficulty faced by the maintainer in order to follow the flow control of the program. It is considered an indicator of the effort needed to understand and test the source code [47]. Kemerer and Slaughter [30] used McCabe's cyclomatic metric to evaluate decision density, which represents the cognitive burden on a programmer in understanding the source code. In order to compute cyclomatic complexity, each source code file was subjected to a commercial software code analysis tool. To account for the effects of size, the complexity metric was normalized by dividing it by the number of lines of code for each software project. This procedure also reduces collinearity problems when size is included in the regression models [20]. The Change in Cognitive Complexity (*ChgCC*) was calculated by subtracting cyclomatic complexity measure of the first version from the cyclomatic complexity measure of the second version, i.e., $CC_{2nd} - CC_{1st}$.

Change in Number of Bugs and Time Taken to Fix Bugs

Various elements of data were extracted from the bug tracking system and the Concurrent Versioning System (CVS) reports, including the bugs reported, the date on which the bugs were reported, the date on which the bugs were fixed, and the version number. One problem faced was that all the bugs in the current version were not closed at the time of the study. To overcome the problem, earlier versions that had more than 90% of the bugs closed at the time of study were included. From these extracted elements, the number of bugs reported and the time taken to fix them for different software versions were computed. From the number of bugs and the time to fix these bugs for each version, the change in the number of bugs (*ChgBugsReported*) over previous version and the change in the average time to fix the bugs (*ChgFixTime*) were computed (*i.e.,* $BugsReported_{3rd}$ - $BugsReported_{2nd}$).

Contributions from New Developers

Software developers use CVS to manage the software development process. CVS stores the current version(s) of the project and its history. A developer can check-out the complete copy of the code, work on this copy and then check back the changes. The modifications are peer reviewed ensuring quality. CVS updates the modified file automatically and registers it as a commit. CVS keeps track of what change was made, who made the change, and when the change was made. This information can be gathered from the log files of the CVS repository of a project. As CVS commits provide a measure of novel invention that is internally validated by peers [10][23], the number of CVS commits is used as a measure of contributions of developers. A commit is considered as 'contribution from a new developer', when the developer has not contributed to the previous version. The number of contributions made by new developers is represented as *ChgNewDevs* (*i.e.,* $ChgNewDevs_{3rd}$ - $ChgNewDevs_{2nd}$).

Control Variables

Age

Brook's Law [7] states that "adding more programmers to a late project makes it later". Based on this, adding new developers at later stages will increase the average time taken to fix bugs. On the other hand, age may indicate the legitimacy and popularity of the software. Popular software attracts more developers and thus older software will have higher number of contributions from developers. To control for age, the *Age* variable is defined as the number of months till the second release since a project's inception at SourceForge.

Size

Size is the oldest measure of software complexity and is believed to be a major driver of software maintenance effort [53]. Larger software is likely to receive more enhancements and more repairs than smaller software, *ceteris paribus*, as larger software embodies greater amount of functionality subject to change. The larger the software, the more difficult it is to test and validate its functionality. This implies that larger software tend to incorporate more errors. Keeping the above in mind, *Size* is used as a control variable and is captured by the number of lines of code of the second release.

Number of downloads

OSS developers can leverage the law of large numbers to identify and fix the bugs [41]. Given enough eyeballs, all bugs are shallow. A huge user base for the software implies that the software will be tested in numerous different environments, more bugs will surface, these will be communicated efficiently to more bug fixers, the fix being obvious to someone, and the fix will be communicated effectively back and integrated into the core of the product. To isolate this effect, the number of cumulative downloads (*Downloads*) of till second release of the project is used as a control variable.

New Developer Knowledge and Skills

The literature on performance has identified individual characteristics such as knowledge and skills as antecedents. Such characteristics are, however, difficult to measure, and are frequently measured through the use of surrogate measures like the level of education and experience. Curtis et al. [11] reported that in a series of experiments involving professional programmers, the number of years of experience was not a significant predictor of comprehension, debugging, or modification time, but that number of languages known was. They suggest that the breadth of experience may be a more reliable guide to ability than length of programming experience. In this work, we also use the breadth of the experience as a surrogate for developer's knowledge and skills. So, to control for the effect of new developers' skills, the variable *SkillsChg* (*i.e.* $Skills_{2nd}$ –$Skills_{1st}$) was used and was measured by the change in team skills with the addition of new developers to the team.

Sponsorship

An increasing number of open source projects have opted to receive monetary donations from organizations and users. Although some developers and projects choose to allocate part or all of the incoming donations to SourceForge, most recipients of the donations rely on monetary support to fund development time and other key resources that are necessary for the continuation of the projects. It is expected that developers receiving additional monetary benefits will devote extra effort and time into comprehending and fixing the source code. The control variable *AcceptSponsors* is used to capture whether a project is accepting external funds and using monetary compensation as part of its incentive mechanism. It takes the value of 1 if the project is accepting donations and 0 otherwise.

Development Status and Maturity

To capture the development stage of a project, which is typically determined by the developer in charge of the project on SourceForge, the control variable *DevStatus* takes values ranging from 1 to 6 representing development stages of Planning, Pre-Alpha, Alpha, Beta, Production/Stable, and mature respectively. *DevStatus* was also measured at second release. The larger the value of *DevStatus*, the more mature the project is.

Transformations

Initial investigations indicated that the dependent variable and many of the independent variables were not normally distributed. In such case, linear regression analysis might yield biased and non interpretable parameter estimates [19]. Therefore, as suggested by Gelman and Hill [19], a logarithmic transformation on the dependent and the not-normally distributed independent variables was performed.

TABLE 1 — Regression Results

Model	Hypothesis 1		Hypothesis 2		Hypothesis 3		Collinearity Statistics (VIF)
	β	Sig.	β	Sig.	β	Sig.	
ChgCC	.303	.000	-.359	.000	.720	.000	1.150
Size	.228	.000	-.157	.000	.010	.775	1.160
Downloads	.173	.000	.099	.016	-.100	.004	1.205
AcceptSponsors	-.171	.000	.330	.000	-.082	.012	1.053
DevStatus	-.067	.083	.097	.011	-.010	.764	1.042
Age	.156	.000	.038	.350	-.040	.245	1.177
SkillsChg	-.016	.664	-.105	.006	.069	.030	1.014
Adjusted R-Square	0.375		0.385		0.561		

RESULTS

The Variance Inflation Factor (VIF) was computed for all variables in order to test for multicollinearity. VIF is one measure of the effect other independent variables might have on the variance of a regression coefficient. Large VIF values indicate high multicollinearity. Studenmund [50] recommends a cut of 10 for VIF. The VIF values for the different variables in the regression analyses are reported in Table 1, and in no case exceed 1.2. The low VIF values indicate that multicollinearity is not a serious problem.

As we are interested in studying the impact of change of complexity on three dependent variables which are largely distinct, we formulate three separate regression equations analyzing each of the dependent variables. For the dependent measure, *ChgBugsReported*, the impact of change in complexity on the number of bugs (Hypothesis H1) was found by estimating the parameters in the following regression model:

$$ChgBugsReported = \alpha + \beta_1 ChgCC + \beta_2 lnSize + \beta_3 lnDownloads + \beta_4 AcceptSponsors + \beta_5 DevStatus + \beta_6 lnAge + \beta_7 lnSkillsChg$$

A positive and significant estimate of parameter β_1 would indicate that the probability of having bugs in a source code increases as the cognitive complexity of software increases. The results of the regression (Hypothesis 1) are presented in Table 1. The model shows a good fit with the data ($F=33.552$, $p<0.00$). The parameter estimate for *ChgCC* is positive and significant ($\beta_1=0.303$, $p<0.00$). The results suggest that projects with unit increase in cognitive complexity experience 0.303 units increase in the number of bugs, and H1 is supported. The studied variables explained 37.5% of the total variance in the change in bugs reported ($R^2=0.375$).

Tested next is the impact of complexity on the number of contributions from new developers (hypothesis H2) by estimating the parameters for the following regression model:

$$ChgNewDevCommits = \alpha + \beta_1 ChgCC + \beta_2 lnSize + \beta_3 lnDownloads + \beta_4 Sponsors + \beta_5 DevStatus + \beta_6 lnAge + \beta_7 lnSkillsChg$$

The results of the regression (Hypothesis 2) are presented in Table 1. The model shows good fit with the data ($F=34.702$, $p<0.000$). The parameter estimate for *ChgCC* is significantly negative ($\beta_1=-0.359$, $p<0.000$). The results suggest that a unit increase in cognitive complexity decreases the contributions from new developers by 0.359 units. Hypothesis H2 is supported. The studied variables explained 38.5% of the total variance in the change in new developers'' commits ($R^2=0.385$).

Finally, examined is the impact of complexity on the time taken to fix bugs (hypothesis H3) by estimating the parameters for the following regression model:

$$Time\ to\ fix\ bugs = \alpha + \beta_1 ChgCC + \beta_2 lnSize + \beta_3 lnDownloads ++ \beta_4 Sponsors + \beta_5 DevStatus + \beta_6 lnAge + \beta_7 lnSkillsChg$$

Table 1 shows the results of the regression analysis (Hypothesis 3). The model shows a good fit with the data ($F=70.660$, $p<0.000$). The parameter estimate for *ChgCC* is significant and positive ($\beta_1=0.720$, $p<0.000$), indicating that projects that experience a unit increase in cognitive complexity takes 0.720 units additional time to fix bugs. Thus hypothesis H3 supported. The studied variables explained 56.1% of the total variance in the change in time taken to fix the reported bugs ($R^2=0.561$).

DISCUSSION AND IMPLICATIONS

Main Effects

The increase in the cognitive complexity of open software as it evolves over time is of significant concern, as it will make software maintenance increasingly difficult. In the extreme, developers may stop making fixes and refinements rendering the software error-prone and obsolete. Ultimately the open software may perish its own death, be replaced by another software project, or may go a major and laborious overhaul; all options are expensive. In this section, we discuss our findings on how complexity and control variables influence different aspects of software maintenance.

The literature shows mixed support for the negative impact of complexity on software quality. For example, Harter and Slaughter [25] found a negative association between complexity and quality. However, Gaffney [18] did not find software complexity to be associated with error rates. Fitzsimmons and Love [16] reported that the correlation between cognitive complexity and the reported number of bugs ranges from 0.75 to 0.81. In our data, the correlation between the number of bugs reported and complexity was 0.43. It is interesting to note that the correlation found in this study was much smaller than the correlations reported in earlier studies for non-open source software; however, it is consistent with the literature on OSS. In the context of OSS, Schröter et al. [44] reported the correlation value in the range of 0.40. Furthermore, Kemerer and Slaughter [30] found that complex software is more frequently repaired, which has the effect of increasing the number of bugs. Therefore, it can be said with confidence that as the complexity of the software increases, the number of reported bugs, and by implication the actual number of bugs increases.

Another measure of software quality is the time taken to fix bugs. In fact, by mining software histories of two projects, Kim and Whitehead [32] recommended to use time taken to fix bugs as a measure of software quality. In our analysis, we found that the complexity of software has a strong positive influence on the time taken to fix bugs. It is common that when a bug is fixed in one segment of the source code, it usually causes ripple effects and adjustments in other segments [36]. The more complex the software is, the more are the adjustments in other segments. As a consequence, the developer has to simultaneously understand, and repair related pieces in dispersed segments. Handling all segments together has a detrimental effect on the time devoted by the developer because more time is needed to follow the flow of logic within the code [3]. This is supported by several empirical studies that have found that time required to fix bugs increases as complexity increases [5][20]. This result has another spurious effect on software maintenance. When a developer becomes conscious of long time needed to fix a bug, there is tendency for the developer find "quick and dirty" solutions, thereby making the code even less maintainable. Such half-baked efforts lead to a vicious cycle in which the complexity, the number of bugs, and the time taken to fix those bugs feed on each other until a dead end is reached with the only option of either reengineering the project or shutting it down completely.

Another reason for the longer time to fix bugs in complex code can be found in Dymo's [13] observations. Dymo noted that most people prefer to work on software enhancements by adding features rather than working on fixing bugs. This is especially true, when the source code is more complex. Debugging and understanding the existing code, written by someone else, takes more time and resources. As the majority of the work is done on voluntary basis in open software and developers are not bound by contracts, developers tend to work on new versions of the software rather than continue to work on improving the old ones. Although this has the potential of bringing them more visibility in the OSS community, the net effect is further delay in fixing bugs.

Another impact of source code complexity analyzed in the study is on attracting contributions from new developers. Analysis shows that cognitive complexity has a strong negative influence on the number of contributions from new developers. As OSS thrives upon voluntary contributions, the project managers must actively control the source code complexity in order to attract contributions from new developers. In a complex piece of code, it takes longer for a developer to determine the flow of logic resulting in slower progress of the project [40]. Cavalier [8] pointed that the willingness of people to continue to contribute to a project is related to the progress that is made in the project. If a large number of activities do not seem to be moving forward, participants lose interest, leading them to leave the project. This leads to a higher likelihood of activities not being completed, and ultimately, the death of the project. Such projects become inactive over time and fail to attract any contributions.

Effects of Control Variables

Interesting observations can be made based on the effects of the control variables. Our analysis found strong effects of size on the number of bugs and the number of contributions from new developers. It is often argued that complexity and size are strongly correlated and that could lead to the problem of multicollinearity, which tends to inflate regression coefficients. As mentioned earlier, multicollinearity was tested by computing variance inflation factors and was found to be within permissible limits. Accordingly the effects of size are independent of the effects of complexity.

The number of downloads has strong effects on the number of bugs, time to fix bugs, and the number of contributions from new developers. The number of downloads indicates the popularity of a project; popular projects attract more user and developers [33]. As the number of users and developer community grow, the number of eyes watching the source code increases. As Eric Raymond [41] repeatedly mentions "to many eyes, all bugs are shallow". When source code is open and freely visible, users can readily identify flaws. The probability of finding a bug increases with the increase in the number of eyes. As a result, the number of hands working on code also increases leading to increased contributions from new developers.

The continued development of a project, represented by its age, gives software legitimacy, reputation and attention of the community. However, in our study, age did not show any significant effect. The reason could be because a large number of OSS projects on SourceForge are in early stages of development and there was not much variance in the data. This could be attributed to the ease with which new projects can be started. Such projects become inactive over time and have almost zero contributions from the developer community. It could be argued that age can bring legitimacy, reputation, and attention only if the project is active. Therefore, a more reliable indicator of continued development is the development status of a project, which was also studied and was found to have a significant positive impact on the number of commits from new developers. In the OSS literature, development status has been shown to have a positive impact on project's popularity. Al Marzouq et al. [1] argue that a project attracts more developers as the software becomes more stable. In turn, these new developers bring effort and contribution that improves the software. A growth cycle begins a network effect that feeds both the community and development of the software.

Lakhani and Wolf [34] showed that developers receiving money in any form spend more time working on OSS than their peers. Similar results are shown by this study. We found that the projects that have any form of sponsorship have higher number of contributions from new developers. Such projects also had less number of bugs and took lesser time to fix the bugs. This clearly indicates that developers are receptive to external stimuli such as a monetary reward. Henkel [26] illustrated a similar impact of external sponsorship on the development of applications for Linux, one of the most successful OSS project. Henkel noticed that most contributors in the field of embedded Linux are salaried or contract developers working for commercial firms.

The change in team skills with the addition of new developers was found to have significant influence on the number of contributions from new developers and the time taken to fix bugs. However, both relationships were in a direction opposite to what was expected. The expectation was that as new developers increase, the number of contributions will increase and the time taken to fix bugs will reduce. The opposite directions of the relationships indicate that with the increase in number of skills, the overall time to fix bugs increases and the new contributions decrease. A logical explanation is that either the developers are just joining the development team without actually contributing towards project development or the amount of contributions is not proportionate to the number of skills they possess. Possibly the same core group of developers are largely responsible for the majority of contributions, and new developers do not add anything substantive. This logic is consistent with the commonly held belief in OSS that development follows Pareto's law, where a small number of developers (~20%) are responsible for the majority of the work accomplished (~80%).

LIMITATIONS AND CONTRIBUTIONS

Some limitations of the study need to be pointed out. The first limitation is the sample frame. While SourceForge has data about a vast collection of OSS projects, it does not capture *all* OSS projects, which is the ultimate population of interest. While the sample size is by far large enough to ensure statistical validity, the choice of the sample frame may have some bearing on the outcomes of the study. Additionally, it can be argued that the change log only records the committer; whether the developer of the code is ever acknowledged is uncertain. And, do all bugs get reported? There could be bugs that are probably fixed but never reported.

In spite of the limitations, this study makes important contributions to both the literature and practice. The results are robust as the hypotheses regarding cognitive complexity were supported after having controlled for various factors. In other words, our conclusions cannot be seen as artificial due to possible correlation with other factors. The most important contribution is the strong support for the relationships between cognitive complexity and software quality, and cognitive complexity and contributions from new developers. Our models indicate that, on the average, OSS development projects with high cognitive complexity are significantly associated with increased bugs and repair time and decreased contributions from new developers. These findings have at least two immediate implications for software managers and project administrators. First, they must measure software complexity on a continual basis, at least once for each release or at regular intervals. Second, they need to implement guidelines for upper bounds of complexity and recommend that software versions at no stage exceed these guidelines. However, no standard guidelines are probably universally applicable for all software development projects. Developers and administrators may want to set their own standards for their specific projects, like the NSA (National Security Agency) standard, which is derived from an analysis of 25 million lines of software code written for NSA.

Furthermore, project administrators for OSS projects need to learn the importance of controlling complexity. As recommended by Lehman [35], strategies need to be developed not only to control complexity, but also to actively reduce it. As a software project progresses, it becomes increasingly complex making it difficult to understand and manage [14]. Project administrators need to be careful about subsequent changes between different versions. Such changes can have strong debilitating impacts on projects. If changes are not well monitored, they can lead to a ripple effect. Ripple effect refers to the phenomenon of changes made to one part of the software affecting and propagating to other parts of the software. Lehman's operating system example clearly shows the ripple effect since the percentage of modules changed in Release 15 is 33% while the percentage of modules changed in Release 19 is 56%. The OSS development, thriving on voluntary contributions, must keep a close watch on the cognitive complexity of the software in order to attract contributions from new developers.

Another important contribution of this research is for organizations involved in or interested in getting involved in OSS development. Our results indicate that, contrary to OSS ideological beliefs, offering a monetary reward for participation may successfully attract increased contributions from the OSS community.

REFERENCES

[1]. AlMarzouq, M., Zheng, L., Rong, G., and Grover, V. "Open Source: Concepts, Benefits, and Challenges," *Communications of the AIS*, 16, Article-37, 2005, pp.756:784.

[2]. Banker, R. D., Datar, S., Kemerer, C., and Zweig, D. "Software Complexity and Maintenance Costs," *Communications of the ACM*, 36(11), 1993, pp.81-94.

[3]. Banker, R., Davis, G., and Slaughter, S. "Software development practices, software complexity, and software maintenance effort: a field study," *Management Science*, 44(4), 1998, pp.433-450.

[4]. Basili, V. and Hutchens, D. "An Empirical Study of a Syntactic Complexity Family," *IEEE Trans. Software Engineering*, 9, 1983, pp.664-672.

[5]. Boehm, B. *Software Engineering Economics*, Prentice-Hall, New York, 1981.

[6]. Bonaccorsi, A., and Rossi, C. "Why open source software can succeed," *Research Policy*, 32(7), 2003, pp.1243-1258

[7]. Brooks, F. *The Mythical Man-Month*, Addison-Wesley, Reading, Mass., 1975.

[8]. Carillo, K. and Okuli, C. "The Open Source Movement: A Revolution in Software Development," *Journal of Computer Information Systems,* 49(2), Winter2008/2009, pp.1-9.

[9]. Cavalier, F. "*Some Implications of Bazaar Size*," 1998, available at http://www.mibsoftware.com/bazdev/ accessed 8 May 2006.

[10]. Crowston, K., Annabi, H, Howison, J. "Defining Open Source Software Project Success," *Proceedings of ICIS*, Seattle, WA, 2003.

[11]. Curtis, B., Sheppard, S., Milliman, P., Borst, M., and Love, T. "Measuring the Psychological Complexity of Software Maintenance Tasks with the Halstead and McCabe Metrics," *IEEE Transactions on Software Engineering*, 5(2), 1979, pp.96-104.

[12]. Darcy, D., Kemerer, C., Slaughter, S., and Tomayko, J. "The Structural Complexity of Software: An Experimental Test," *IEEE Transactions on Software Engineering*, 31(11), 2005, pp.982-995.

[13]. Dymo, A. "Open Source Software Engineering," *II Open Source World Conference, Málaga,* 2006.

[14]. Feller, J. and Fitzgerald, B. *Understanding open source software development*, London: Addison-Wesley, 2002.

[15]. Fitzgerald, B. "Has Open Source Software a Future?,"

Perspectives on Free and Open Source Software, MIT Press, 2005, pp.93-106.

[16]. Fitzsimmons, A. and Love, T. "A review and evaluation of software science," *Computer Survey,* 10(1), 1978, pp.3-18.

[17]. Fjeldstad, R. and Hamlen, W. "Application program maintenance-report to our respondents," *Tutorial on Software Maintenance*, 1983, pp. 13-27.

[18]. Gaffney, J. "Estimating the Number of Faults in Code," *IEEE Transactions on Software Engineering*, 10(4), 1984, pp. 13-27.

[19]. Gelman, A., and Hill, J. *Data Analysis Using Regression and Multilevel/Hierarchical Models*, Cambridge University Press, 2007.

[20]. Gill, G. and Kemerer, C. "Cyclomatic complexity density and software maintenance productivity," *Transactions on Software Engineering*, 17(12), 1991, pp. 1284-1288.

[21]. González-Barahona, J., Miguel A, Pérez, O, Quirós, P., González, J., and Olivera, V. "Counting potatoes. The size of Debian 2.2," *Upgrade*, 2(6), 2001, pp. 60-66.

[22]. Gorla, N., and Ramakrishnan, R. "Effect of Software Structure Attributes Software Development Productivity," *Journal of Systems and Software*, 36(2), 1997, pp. 191-199.

[23]. Grewal, R., Lilien, G., Mallapragada, G. "Location, Location, Location: How Network Embeddedness Affects Project Success in Open Source Systems," *Management Science* 52(7), 2006, pp. 1043-1056.

[24]. Harrison, W. and Cook, C. "Insights on improving the maintenance process through software measurement," *Proceedings of Conference on Software Maintenance*, San Diego,CA, 1990, pp. 37-44.

[25]. Harter, D. and Slaughter, S. "Process maturity and software quality: a field study," *International Conference on Information Systems,* Brisbane, Australia, 2000, pp. 407-411.

[26]. Henkel, J. "Selective Revealing in Open Innovation Processes: The Case of Embedded Linux," *Research Policy*, 35(7), 2006, pp. 953-969.

[27]. Henry, S., Kafura, D., and Harris, K. "On the Relationship among Three Software Metrics," *ACM SIGMETRICS: Performance Evaluation Review*, 10(1), 1981, pp. 81-88.

[28]. Jones, T. *Programming Productivity*, McGraw-Hill, Inc., New York, 1986.

[29]. Kearney, J., Sedlmeyer, R., Thompson, W., Gray, M., and Adler, M. "Software Complexity Measurement," *Communications of the ACM*, 29(11), 1986, pp. 1044-1050.

[30]. Kemerer, C. and. Slaughter, S. "Determinants Of Software Maintenance Profiles: An Empirical Investigation," *Software Maintenance: Research And Practice*, 9(4), 1997, pp. 235-251.

[31]. Kemerer, C. F. "Software complexity and software maintenance: A survey of empirical research," *Annals of Software Engineering,* 1(1), 1995, pp. 1-22.

[32]. Kim, S., Whitehead, E, and Bevan, J. "Analysis of signature change patterns," *Proceedings of the 2005 international workshop on Mining software repositories*, St.Louis,MO, 2005, pp. 1-5.

[33]. Krishnamurthy, S. "Cave or Community? An Empirical Examination of 100 Mature Open Source Projects," *First Monday*, 7(6), 2002.

[34]. Lakhani, K., and Wolf, B. "Why Hackers Do What They Do: Understanding Motivation and Effort in Free/Open Source Software Projects," *Perspectives on Free and Open Source Software,* MIT Press, Cambridge, 2005.

[35]. Lerner, J., and Tirole, J. "Some Simple Economics of Open Source," *The Journal of Industrial Economics*, 1(2), 2002, pp. 197-234.

[36]. Loch, C., Mihm, J., and Huchzermeier, A. "Concurrent Engineering and Design Oscillations in Complex Engineering Projects," *Concurrent Engineering*, 11(3), 2003, pp. 187-199.

[37]. Markus, M., Manville, B., and Agres, C. "What makes a virtual organization work?," *Sloan Management Review,* 42(1), 2000, pp. 13-26.

[38]. Opensource.org, "The Open Source Definition (Version 1.9)", 2002, at http://www.opensource.org/ docs/definition.html, accessed 5 May 2006.

[39]. Pigoski, T. *Practical Software Maintenance.* Wiley computer publishing, 1997.

[40]. Ramanujan, S. and Cooper, R. "A human information processing approach to software maintenance," *Omega*, 22(2), 1994, pp. 85-203.

[41]. Raymond, E. "The Cathedral and the Bazaar," 1999, at http://tuxedo.org/~esr/writings/cathedral-bazaar/

[42]. Raymond, E. *The cathedral and the bazaar: musings on Linux and open source by an accidental revolutionary,* Sebastopol, CA, O'Reilly, 2001.

[43]. Rilling, J. and Klemola, T. "Identifying Comprehension Bottlenecks Using Program Slicing and Cognitive Complexity Metrics," *Proceedings of the 11th IEEE International Workshop on Program Comprehension,* 2003, pp. 115.

[44]. Schröter, A., Zimmermann, T., Premraj, R., and Zeller, A. "If Your Bug Database Could Talk . . . ," *Proceedings of ACM-IEEE 5th International Symposium on Empirical Software Engineering*, *Volume II: Short Papers and Posters,* Brazil, 2006.

[45]. Sen, R, Subramaniam, C, and Nelson, M. "Determinants of the Choice of Open Source Software License," *Journal of Management Information Systems*, 25(3), 2008-9, pp. 207-240.

[46]. Smith, N., Capiluppi, A., and Ramil, J. "Agent-based Simulation of Open Source Evolution," *Software Process Improvement and Practice*, 11(4), 2006, pp. 423-434.

[47]. Stamelos, I.; Angelis, L.; Oikonomou, A.; and Bleris, G. "Code Quality Analysis in Open Source Software Development," *Information Systems Journal*, 12(1), 2002, pp. 43-60.

[48]. Stewart, K., Ammeter, A., Maruping, L. "A Preliminary Analysis of the Influences of Licensing and Organizational Sponsorship on Success in Open Source Projects," *Proceedings of the 38th Hawaii International Conference on System Sciences,* 2005, pp. 197-203.

[49]. Stewart, K., Darcy, D., Daniel, S. "Observations on Patterns of Development in Open Source Software Projects, Open Source Application Spaces," *Fifth Workshop on Open Source Software Engineering,* 2005, St Louis, MO, pp. 1-5.

[50]. Studenmund, A. *Using Econometrics: A Practical Guide*, Harper Collins, New York, NY, 1992.

[51]. vonHippel, E., G. vonKrogh. "Open Source Software and the "Private-Collective" Innovation Model: Issues for Organization Science," *Organization Science,* 14(2), 2003,

pp. 209-225.

[52]. Weyuker, E. "Evaluating software complexity measures," *IEEE Transactions on Software Engineering*, 14(9), 1988, pp. 1357-1365.

[53]. Withrow, C. "Error Density and Size in Ada Software," *IEEE Software,* 7(1), 1990, pp. 26-30.

[54]. Xu, J., Gao, Y, S. Christley, G. Madey. "A Topological Analysis of the Open Source Software Development Community," *Proceedings of the 38th HICSS,* 2005, pp. 198.