Follow this and additional works at: http://aisel.aisnet.org/amcis2007

Recommended Citation
http://aisel.aisnet.org/amcis2007/25

This material is brought to you by the Americas Conference on Information Systems (AMCIS) at AIS Electronic Library (AISeL). It has been accepted for inclusion in AMCIS 2007 Proceedings by an authorized administrator of AIS Electronic Library (AISeL). For more information, please contact elibrary@aisnet.org.
Retention and Quality in Open Source Software Projects

ABSTRACT
Open source software (OSS) is a rapidly developing phenomenon and is finding an increasing use worldwide. In spite of its attractiveness and advantages, issues related to the quality of the software and retention of developers persists. In this study, we identify two key antecedents: complexity and modularity of the software and investigate their effects on software quality and developer retention. We found that both the higher modularity translates into higher developer retention and lesser time to fix bugs; and complexity translates into lower developer retention and more time to fix bugs. In total, four of the six proposed hypotheses are supported by our research.

INTRODUCTION
Over the last two decades, a new software programming and distribution practice has emerged, which when combined with novel uses of intellectual property law, has come to be known as “open source software” (OSS) development. It differs from other forms and practices of software development in several respects, the most important being the availability of source code to a consumer.

A typical open source project starts when an individual (or small group) feels a need for a new feature or an entirely new software, and someone among that group, eventually writes one. In order to share it with others who have similar needs, the software is released under a license that allows the community not only to use but also to see the source code and modify it to meet local needs and improve the product by fixing bugs. This new community of users and developers of the software may then contribute their changes back to the community. This results in a software that is said to evolve more quickly and meet user needs better than commercial closed-source packages.

It is important to note that these Open Source Software development projects thrive mainly upon the free contributions from its developer community (Stewart et al 2005, Feller and Fitzgerald 2001, Siedlok 2002). Hence, it becomes extremely important for those projects to retain the existing developer community and attract new developers.

Motivation has been identified in the OSS literature as a way of attracting and retaining developers (Lakhani et al Bates 2002; Hertel et al 2003; Bonaccorsi and Rossi 2003, Marcus, et al, 2000). Motivation can only lead to making developers attempt to contribute. Whereas, the actual ability to contribute will be limited by the developer’s comprehension of the existing software source code. As Cavalier (1998) points out, “Willingness of people to continue to contribute is related to the progress that is made. If a large number of activities do not seem to be moving forward, participants will lose interest or bicker thus reducing effective size and power. This leads to a higher likelihood of activities not being completed, and ultimately, the death of the project”. This loss in interest could eventually lead to developers leaving the project.
It becomes extremely important for OSS developers and administrators to ensure that the source code is well written and can be understood by other developers. Two such factors that could affect the quality of code are: the complexity and the modularity of software. In this paper, we focus on both complexity and modularity of software and its influence on a software project. The analysis we found suggests that software’s source code complexity and modularity have a strong influence on retaining developers and the number of bugs in a software project.

LITERATURE & MODEL DEVELOPMENT
Software complexity has existed as an important issue ever since the software programs came into existence and it is strongly connected to the amount of resources needed in a project. This has been stated by most of the researchers in software metrics (Jones, 1996; Fenton & Pfleeger, 1996). The notion is that a more complex problem or solution demands more resources, in terms of man-hours, computer time, support software, etc, from the project. In complex software, a large share of the resources is used to find errors, debug, and retest; thus, associated measures of the consequences of complexity are the number of software bugs, average time taken to fix those bugs, and the retention rate of developers.

Complexity
The notion of structural complexity can also be seen from different viewpoints, each playing a different role. We can identify at least three different aspects of structure (Fenton & Pfleeger, 1996): control-flow structure, information-flow structure, and data structure.

“Control-flow” is concerned with the sequence in which instructions are executed in a program. This aspect of structure takes into consideration the iterative and looping nature of a program. Thus, understanding control flows make more visible the fact that an instruction may be executed many times as the program is actually run.

“Information-flow”, also referred to as data-flow, concerns the trail of a data item as it is created or handled by a program. Many times, the transactions applied to data are more complex than the instructions that implement them; data-flow measures depict the behavior of the data as it interacts with the program.

“Data-structure” is the organization of the data itself, independent of the program. When data elements are arranged as lists, queues, stacks, or other well-defined structures, the algorithms for creating, modifying, or deleting them are more likely to be well-defined, too. So the structure of the data tells us a great deal about the difficulty involved in writing programs to handle the data, and in defining test cases for verifying that the programs are correct. Sometimes a program is complex due to a complex data structure rather than complex control or data flow.

Effects of software complexity
The reason as to why we are concerned about complexity is that it is related in one way or the other to the more important attributes of the software process. But how do these relationships look more in detail? This is the subject for the following section. According to our view, software complexity is a determining factor for three of the main attributes of the software product: number of bugs found, time taken to fix those bugs, and retaining the developers contributing to a project.
**Number of Bugs**

A defect is a result of an error during the software development process (Fenton and Pfleeger 1996). The error occurs because of the human factor, often of ignorance, incompetence or negligence. Error-prone modules are the modules that are most likely, statistically, to have a high proportion of errors (Fenton and Pfleeger 1996). The idea of identifying error-prone modules is to have the possibilities and the time to correct the errors and prevent them from resulting in faults and failures.

The main idea behind the relationship between complexity and error-proneness is that when comparing two different solutions to the same computational problem provided that all other things are equal, the most complex solution generates the most number of errors. 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 (Curtis et al 1979; Henry et al 1981).

This confirmation of the existence of a relationship between software complexity and errors/bugs, and understanding the characteristics of the relationship, is of important practical benefit. Since a great deal of software development costs are directed to software maintenance and testing, it is crucial for a project to have the indicators for predicting and identifying the type of errors that may occur in a specific module (Basili & Weiss, 1984; Boehm, 1981).

This leads us to our first hypothesis:

\[
H1: \text{OSS projects with higher complexity have more errors/bugs in the software than OSS projects with lower complexity}
\]

**Time to Fix Bugs**

Complexity influences the testability, maintainability, and the understanding of the software product. Maintainability is a requirement when we want our software to be easy to understand, enhance, or correct. A complex product, in general, also demands a larger share of resources to modify and correct.

When we create programs that are easy to maintain, it is also easier to detect and correct our mistakes, and thereby reach our quality goals. Further, a maintenance friendly module does not need as much resources or turn around time to fix repairs, thus enhancing the project productivity. Thus our next hypothesis:

\[
H2: \text{OSS projects with higher complexity will take more time to fix bugs than OSS projects with lower complexity.}
\]

**Retaining Developers**

Since the human mind is limited, the size/complexity of the software also influences the ability of programmers and system developers to understand and comprehend how the software is
structured. A complex unit of code is more time-consuming and resource-demanding to familiarize oneself with, and it is even possible that it is too complex to be able to comprehend at all. Thus, added unnecessary complexity in large software modules requires additional resources that could be used for other activities, and thereby it lowers the productivity of the project.

In such situations, where OSS developers seek to gain personal satisfaction and value from peer reviews, their inability to contribute appropriately could lead them to leave the project. This leads us to our next hypothesis:

**H3:** OSS projects with higher complexity will have lower retention of developers than software projects with lower complexity

**Modularity**
Brooks' (1975) recipe for coping with the design and the production of complex software was to divide labor into separate possible high level activities from lower ones. This approach aims at reducing the degree of interdependencies by decomposing a complex project into smaller and independent subparts that are loosely coupled and highly independent of each other (von Hippel 1990; Langlois 2002). Hence, when subparts are almost independent it is possible to divide labor, minimizing coordination and communication costs. By its ability to easily substitute old designs with new ones at low cost, modularity accommodates future uncertainty (Baldwin and Clark 1997) as well.

Modularity is an extremely critical component in OSS development. As mentioned by Torvalds (1999) about the concept of modularity, "for without it, you cannot have people working in parallel". Modularity means that the kernel itself and plans for its future development is organized around small, manageable pieces.

With a modular design, multiple programmers (perhaps unknown to one another) can be working to build new functions into the same module. This parallel approach is thought to spur innovation and can lead to a rapid development process. Modularity also allows development to continue but avoids a situation where the impact of one person's enhancements to a module leads to problems with the work in some other module. Furthermore, modularity enables the project content manager to keep better control as work progresses and as the product gets more complex (DiBona et al 1999).

Advocates of OSS argue that having a large team means that OSS is by necessity ‘modular’ (made up of discrete units, each with a specific function). Modularity simplifies software design and can increase the reliability as well as flexibility of software. High modular nature of OSS allows developers to carry out development of specific parts of the system with autonomy and without any need to coordinate their efforts with other sub-projects. Modularity not only allows parallel development but also contribution of new components and modules allowing the substantial improvement of the overall design of the system via module innovation and competition between similar projects (both completely new modules and variation and improvements in existing ones) (Narduzzo and Rossiy 2003).
As increased modularity helps in the designer’s understanding of the problem (Lew et al., 1988), it increases the chances of the designer’s contribution to the project problem. As a result, the overall developers’ contribution to a project increases. Hence, our next set of hypothesis:

- **H4**: OSS projects with high modularity will have more errors/bugs in the software than OSS projects with lower modularity.
- **H5**: OSS projects with higher modularity will take lesser time to fix bugs than OSS projects with lower modularity.
- **H6**: OSS projects with higher modularity will have higher retention of developers than OSS projects with lower modularity.

**DATA COLLECTION AND MEASUREMENTS**

Researchers have examined OSS from different levels of analysis, including viewing OSS as a phenomenon at the community organization to a project level. The focus of this research is on understanding OSS success at the project level. Based on the suggestion of von Hippel and von Krogh (2003), we use SourceForge repository available through www.sf.net to collect publicly available data to test the hypotheses.

We used a stratified random sampling technique to select projects to be included in the study. We first selected three project categories from which to draw our sample, namely, utilities, software development, and games. These categories are chosen as they represent software for different kinds of community and a large numbers of projects. Additionally, to be eligible as a sample, a project must have to be released between Jan 2000 to Dec 2006. After selecting the 70 different software projects, we downloaded their source code files and ran the source codes through CCCC, a metric calculator available on SF, to find the complexity and modularity of each of these software projects. CCCC defines modules in terms of a grouping of member functions - C++ classes and namespaces, Java classes and interfaces, and computes McCabe’s and Fan-in-Fan-out numbers for complexity.

To avoid the bias of the project’s size on the measures of complexity, we used complexity per line of code as a measure to represent the complexity of a project. Time taken to fix a bug was calculated by subtracting the date the bug was reported from the date the bug was fixed within the time frame. The number of developers working on a project was found from the change log files, and the change in developers was calculated by counting the number of developers contributing during a release compared to the number of contributing developers in the previous release.

Note that there are different releases of software projects. We have only considered one release per project for our study. But to calculate the change in number developers, two releases for each software project were studied.

**RESULTS**

We used LISREL to run the regression analysis. The results shown in Figure 1 represent the standardized coefficients.
Our data supports the model, which is indicated by the model fit statistics: GFI= 0.973 and AGFI= 0.866. All our hypotheses are supported other than the two hypotheses dealing with the number of bugs fixed. Our results indicate that complexity and modularity have a strong influence on the number of bugs being reported and the retention of developers (significant up to 0.001). It seems that the number of bugs fixed may either not be a good indicator of the software quality processes or may have different antecedents.

<table>
<thead>
<tr>
<th>Hypothesis</th>
<th>Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>H1</strong> OSS projects with higher complexity have more errors/bugs in the software than OSS projects with lower complexity</td>
<td><strong>No</strong></td>
</tr>
<tr>
<td><strong>H2</strong> OSS projects with higher complexity will take more time to fix bugs than OSS projects with lower complexity</td>
<td><strong>Yes</strong></td>
</tr>
<tr>
<td><strong>H3</strong> OSS projects with higher complexity will have lower retention of developers than software projects with lower complexity</td>
<td><strong>Yes</strong></td>
</tr>
<tr>
<td><strong>H4</strong> OSS projects with high modularity will have more bugs fixed than the OSS projects with low modularity</td>
<td><strong>No</strong></td>
</tr>
<tr>
<td><strong>H5</strong> OSS projects with higher modularity will take lesser time to fix bugs than OSS projects with lower modularity</td>
<td><strong>Yes</strong></td>
</tr>
<tr>
<td><strong>H6</strong> OSS projects with higher modularity will have higher retention of developers than OSS projects with lower modularity</td>
<td><strong>Yes</strong></td>
</tr>
</tbody>
</table>

**CONCLUSIONS**

With the steep rise in the use of open source software in practically all facets of business, it is critical to examine the quality issues in OSS. Furthermore, for the future and survivability of OSS, it is important that a critical mass of programmers and developers associate with such projects. Both issues of quality and retention were examined in this paper. Data from 70 projects in the SourceForge repository was analyzed to investigate relationships with modularity and complexity in OSS projects. Regression analysis supported many of the proposed relationships. Future research will incorporate more factors in understanding these relationships as well as conduct a longitudinal analysis to develop a greater understanding of the underlying processes.
REFERENCES


