Easier Said than Done: An Empirical Investigation of Software Design and Quality in Open Source Software Development

Caryn A. Conley
cconley8@fau.edu

Lee Sproull
lsproull@stern.nyu.edu

Abstract

We empirically examine the relationship between software design modularity and software quality in open source software (OSS) development projects. Conventional wisdom suggests that degree of software modularity affects software quality. An analysis of 203 software releases in 46 OSS projects hosted on SourceForge.net lends support for a more complex relationship between software modularity and software quality than conventional wisdom suggests. We find that software modularity is associated with reduced software complexity, an increased number of static software bugs, and a mixed relationship with the percentage of bugs closed. We do not find empirical evidence supporting any relationship between modularity and other measures of customer satisfaction. In addition to empirically testing the relationship between modularity and quality, we introduce new measures of software modularity and software quality. Implications are developed for the theory of modularity and the practice of software development.

1. Introduction

For years, researchers and practitioners alike have espoused the importance and significance of the relationship between software design and software quality (e.g., [28] [18] [19] [29]). Theoretically, altering the design of a product should directly affect work processes and software quality by influencing the information processing requirements of the software developer and coordination requirements among programmers [9] [13]. One important concept used to characterize software architecture, degree of modularity, has been theorized to be particularly important. In the domain of conventional software development, increasing modularity is theorized to increase software quality and reduce programmatic effort [19] [24]. However, we lack systematic empirical evidence to support these claims [12] [14]. Previous empirical work primarily consists of case studies, providing anecdotal evidence at best as theoretical support. One reason for the lack of empirical tests of this conventional wisdom stems from the difficulty associated with measuring degree of modularity, because accurately assessing the patterns of interdependencies that exist within the software architecture proves challenging.

Community-based Open Source Software (OSS) development is a unique context in which to examine the relationship between software design and software quality. OSS volunteers around the world use the Internet to contribute software, which is then freely available for others to use and alter. Prominent and successful examples of OSS projects include Apache, Linux, and Mozilla Firefox [27] [26].

We study the effects of software design on software quality in the context of community-based OSS development for several reasons. First, community-based OSS development is relatively free of formal organizational influences, so the independent effects of software architecture should be easier to detect and model. Second, as previously mentioned, OSS is a well-established and important phenomenon itself, with many successful projects. Third, the population of OSS projects is sufficiently large and varied that it is possible to compare differences in software quality. Finally, because OSS development is conducted via the Internet, characteristics of the software and software quality can be observed and measured relatively easily and over time. We empirically examine how the design characteristics of the software itself influence software quality in community-based OSS projects over time. We also introduce new measures of modularity and software quality to empirically examine this relationship.

2. Theoretical Perspective

Software modularity is an important concept used to characterize software design, and describes the degree to which components within a product are independent of one another. This degree of component independence should then directly affect task coordination and independence in software development, and subsequently affect development outcomes, such as software quality.[1] We therefore study this characteristic of software design, degree of modularity, and empirically examine the research question: How does degree of software modularity affect software quality in community-based OSS projects?

Modularity describes the degree to which components within software are independent or loosely coupled from one another, yet function as an integrated whole using interfaces [2] [18]. Interfaces "describe in detail how the [components] will interact, including how they will fit together, connect, and communicate" (p. 86, [1]). Modularity is related to the architecture of the software, defined as "the scheme by which the function of the [software] is allocated to [software] components" (p. 240, [25]). Components refer to a "subassembly… or distinct region of the product" (p. 421, [25]). Relationships among individual components are described by coupling, such that "two components are coupled if a change made to one component requires a change to the other component in order for the overall [software] to work correctly" (p. 422, [25]). Software with a low degree of modularity is considered highly integrated, because the functionality is tightly coupled and interdependent, regardless of the number of components.

Theoretically, software modularity should affect software development and software quality. Modularity facilitates task decomposition, which is a strategy for efficiently organizing the work necessary to create the software [21]. More specifically, as software becomes more modular, the process of dividing a larger task into smaller subtasks becomes easier, and tasks become easier to identify. In the context of conventional software development, prior work has claimed that software modularity improves software quality and reduces programmatic effort in both the development and maintenance of centrally-developed and hierarchically-managed software [19] [24]. It therefore seems natural to expect the degree of software modularity to also affect software quality in OSS projects.

From the literature, then, degree of software modularity should be positively related to software quality. Following from the definition of modularity, all other things being equal, more modular software products have more components that are more loosely coupled. The probability that a change to one component will inadvertently affect another component, thus introducing an error, should decrease in more modular software. As the number of components increases in more modular software, functionality becomes more specialized and isolated within each of the components, and each component contains less functionality on average [2] [19] [20] [25]. In addition, smaller and more loosely coupled components foster more rapid problem resolution because problem identification is more likely localized to specific modules [19]. As the ease of problem identification and resolution increases with increasing modularity, a larger number of existing problems should be identified and resolved within the software application, thus increasing software quality in more modular software.

[1] While this paper focuses on the role of modularity in software development, these ideas are also the conventional wisdom in the development of tangible products.

3. Measurement

3.1. Overview

Researchers have noted the difficulty in measuring degree of modularity [12] [14]. We therefore provide a detailed explanation of the measure introduced and compare it to other measures that have been proposed to evaluate degree of software modularity. In addition, because software quality has multiple dimensions, we discuss the measures used to assess two dimensions of software quality in this paper: intrinsic quality and customer satisfaction.

3.2. Software modularity

OSS provides a unique opportunity to more accurately describe software components and their relationships in a single application. Specifically, it is possible to assess the degree of independence, and thus modularity, among software classes by examining function calls among classes in the source code. Several researchers have used this information to measure software modularity at the class or file level [12] [13] [14]. While these measures provide detailed information regarding software functionality, they do not capture whether classes or files communicate via interfaces, a critical concept used to achieve component independence in modular software. Without addressing the types of relationships among classes or files, their measure cannot correctly assess the degree of software modularity. In addition, individual classes or files do not function as modules or components. In practice, programmers use groups of files or classes, called packages in Java, to implement software functionality [4] [8]. Classes within these packages should be interdependent, and classes contained in different packages should be independent. Programmers implement independence among packages using interfaces. Therefore, we argue that measuring the degree of software modularity using packages as the module or component, not classes or files, and capturing the extent to which interfaces are used to achieve independence is a more appropriate assessment of software modularity.

Many people have written applications to extract information about class relationships, and [15] proposed a method for mathematically evaluating this information at the package level as a measure of software design. The measure assesses the degree to which Java programming constructs (e.g. interfaces and concrete classes) are used to achieve independence among packages. Practitioners support and recommend the use of [15]'s measure and the information used to make the calculation to evaluate software design. In fact, several programs have been written to automatically extract relevant metrics from the source code (e.g. JDepend, NDepend, Metrics, VizzAnalyzer) (e.g. [6] [7] [11]).

We use [15]'s calculation as our measure of *degree of modularity*. To do this, we derive various metrics from the source code for each package contained in the application, which are then included in a series of calculations.[2] We extract these metrics from each release using a static source code analysis tool, the Metrics Eclipse plug-in.[3] These include the total number of lines of code (LOC), number of concrete classes, number of abstract classes, afferent coupling, and efferent coupling.[4] Afferent coupling (AC) is a measure of a package's responsibility, and counts the number of other packages within the application that call (depend on) classes within the given package. Efferent coupling (EC) is a measure of a package's independence, and counts the number of other packages within the project that classes within the given package call (depend on). Higher values of AC indicate higher responsibility and higher values of EC indicate lower independence.

Using these metrics, we then calculate each package's abstractness (A), which is the ratio of the number of abstract classes to total classes within a package. A value of zero indicates a package is comprised of only concrete classes, and a value of 1 indicates a package is comprised of only abstract classes. Next we calculate Instability (I), which is the ratio of a package's independence to total independence and responsibility [EC / (EC + AC)]. A value of zero indicates complete stability (changes may be made to other packages without affecting this package) and a value of one indicates complete instability (changes made to other packages will likely directly affect this package). The last step in calculating this measure is to calculate the "distance from the main sequence" (D), which measures the balance between abstractness and instability in a package. It is the perpendicular distance of a package from the idealized line A + I = 1. A value of zero indicates ideal balance (good design), and one indicates complete imbalance (poor design).

In this paper, $D_{release}$ is used as a proxy for source code modularity because it evaluates the type of relationships among software packages. Because the sizes of packages within an application vary, the implications of each individual package's design may not be equal. Therefore, we weight each package's D by the log of the LOC within the package to account for these differences. To calculate $D_{release}$ for each software application, we sum each package's weighted D and then divide by the sum of the log of the LOC of all packages for every package *i* in the release. Finally, we subtract this from one to get $D_{release}$, so that larger values indicate higher degrees of modularity:

$$D_{release} = 1 - \frac{\sum D_i * \log(LOC_i)}{\sum \log(LOC_i)}$$

To the best of our knowledge, no previous research on OSS development has used this measure.

3.3. Software quality

Software quality consists of several dimensions, such as intrinsic quality and customer satisfaction [10]. Intrinsic quality refers to the inherent quality of the software itself (e.g. does the program work properly when running, or does it break or crash unexpectedly?). Customer satisfaction refers to user perceptions of quality (e.g. does the program contain desired features? Does the program work when I use it?). We use two measures to assess intrinsic software quality and three measures to assess customer satisfaction.

First, we introduce two measures of intrinsic software quality not previously reported in the OSS literature: *number of static bugs* and *software complexity*.[5] Bugs in software refer to problems or

[2] We only analyze source code that is included in the core of the application. We therefore exclude obvious plug-ins designed to port the code to other platforms and applications. Code of this nature is generally included in downloadable files and therefore easily identifiable. We also exclude source code used to unit test the application. Generally this code is included in a separate package or separate folder, and common naming conventions (e.g. "Test") facilitate identification.

[3] http://metrics.sourceforge.net/

[4] "Concrete" classes contain the actual programming logic or functionality, while "abstract" classes act as the interfaces between concrete classes.

[5] Both measures are frequently used in software engineering (e.g., [5] [10] [22]). Number of "bugs" or functional defects is a common operationalization of intrinsic quality [10]. Other traditional

errors with the program that prevent it from running properly. Examples include logical flaws, inappropriate use of programming constructs, or memory leaks. Any of these problems reduce software quality. Software complexity is an additional software design characteristic that affects the difficulty associated with problem identification and resolution, testability, and maintenance provisioning over time. Higher software complexity reflects lower software quality. Both measures are derived from objective evaluations of the source code included in each software release.

We first assess the number of bugs or defects in the source code using two static source code analysis tools, FindBugs and PMD.[6] Both tools provide a list of the problems found, and the location of each problem in the source code. The results of these static source code analysis tools are combined because each uses a different algorithm to identify different types of problems or defects within the source code. This measure should be interpreted in a relative sense, as each tool exhibits some degree of false positive and false negative reporting. *Number of static bugs* is therefore a count of the total number of static source code bugs found by both programs for a given software release. This total is then divided by the log of the total LOC in the release to control for software size.

In addition to the number of bugs or defects in a software program, software complexity also reflects the intrinsic quality of the software written. Software complexity describes the complexity of the control flow logic within an application by examining the number of linearly independent paths within the source code. A greater number of logic paths indicate greater complexity. We use McCabe's Cyclomatic Complexity measure, a standard metric used in software engineering, to measure *software complexity* in this paper [16]. We calculate the average software complexity of each release using information from the Metrics Eclipse plug-in. Both intrinsic measures of quality assess the total release code quality.

Second, we use three measures to assess software quality associated with customer satisfaction as previously identified in OSS research as a proxy for software quality and OSS project success [3].[7] Many OSS projects use a specific communication tool, a bug tracker, to encourage users to voluntarily report errors found in the software.[8] Once a bug has been addressed or solved by project developers, the bug report will be "closed." We calculate three measures of software quality using this bug report information. First, we calculate the *number of bugs reported* as the number of bug reports created for a given software release. A greater number of problems reported by users should reflect lower customer satisfaction and thus lower software quality. We assess two additional measures of customer satisfaction: *percentage of bugs closed* and *time to close bugs*.[9] Increasing the percentage of bugs closed and decreasing the time to close bugs should reflect higher customer satisfaction, indicating that when problems are reported they are fixed and closed in a timely fashion. However, the number of bugs reported is a function of the number of people using the software, and may not reflect the inherent quality of the software. Therefore, because projects with smaller communities may experience fewer bug reports because of the smaller user base, fewer bug reports may not necessarily indicate better software quality. We therefore weighted all three bug report measures by the number of downloads for each release to account for differences in community size and number of people reporting bugs, which has not been done previously to our knowledge.[10]

4. Method

4.1. Overview

To examine how software modularity affects software quality in community-based OSS projects, we require detailed longitudinal information about software design and objective measures of software quality. OSS projects hosted on SourceForge.net meet these requirements. People can freely register any OSS project on SourceForge, and subsequently use the suite

measures of intrinsic quality address conformance to stated requirements [10]. Because most OSS products do not implement specific requirements, we do not use this as a measure of software quality.

[6] http://findbugs.sourceforge.net/ and http://pmd.sourceforge.net/

[7] Reports of problems by customers and customer satisfaction with the software are two common operationalizations of customer satisfaction [10]. We do not report direct measures of customer satisfaction in this paper.

[8] Many OSS researchers have used community problem reports or bug reports as a proxy for software quality. However, not all project communities use specific bug tracking tools – some may use email lists or discussion boards to manage reported bugs making it difficult for researchers to filter identified bugs from other communication. Bug reports are also a proxy for run-time errors, as the problems identified with the software often result from errors when using the software.

[9] We calculate the average time to close bugs only for bug reports that have been "closed". This is therefore a conservative measure of time to fix bugs, as some bugs may remain outstanding with very long working periods.

[10] Because we do not have data on the number of downloads per time period for each project, we estimate the number of downloads for each release using an exponential function and the duration of each release. As the project ages and improves, more people are likely to download the software as the software matures and more people learn about the software, so adoption likely follows an exponential function.

of tools available to all registered projects, including a source code repository, project website, developer website, tracker tools, mailing lists, and discussion boards. Common availability and use of these tools across projects reduces tool differences as a source of observed differences in software quality. In addition, SourceForge is one of the most popular platforms for hosting OSS projects, thus yielding a large population of projects from which to draw our sample. Most importantly, SourceForge provides a public record of the history of each project: all records of communication, release history and associated files, source code repositories, artifact tracking, etc., are publicly available and accessible, providing us with our required data of a cross-section of OSS project data over time.

4.2. Sample

While SourceForge hosts over 160,000 thousand registered projects, not all projects exhibit characteristics that correspond to the types of projects of interest. For example, a large fraction of registered projects are estimated or known to be a vanity project or a project in name only, i.e., they have no developers other than the person who registered the project and exhibit no development or communication activity. Of the remaining "real" projects, some are clearly controlled by a commercial organization and its employees, rather than by volunteers, and thus have a commercial incentive and authority to exercise control over the project. In addition, because we examine how software modularity affects software quality, it is important for each project to have created a working program, even if it is very preliminary, to ensure the existence of an initial code base from which project members could work. Generally speaking, OSS projects in later stages of development have distributed at least one version, or release, of the software. It is also important that the project exhibits some minimum degree of development or communication activity among project members to ensure a sufficiently large pool of potential code contributors. The most common programming language used by the projects registered on SourceForge.net are object-oriented programming languages, including Java and C++. In addition, object-oriented language skills are the most common skills reported by SourceForge.net users. We therefore used the following sampling frame to identify an appropriate group of projects from which to sample: projects that do not exhibit obvious corporate or organizational sponsorship or involvement, projects with at least one software release in a relatively mature development stage (Beta or Production/Stable), projects exhibiting a minimum level of project activity, and products written only using Java.[11] The frame includes approximately 180 projects from which a random sample of 46 was drawn. For each project, we sample each major software release (e.g. 1.x, 2.x) as our unit of analysis for a total of 203 releases. Refer to Table 1 for characteristics of the projects sampled.

Table 1. Summary project characteristics

	Average	*Std Dev*	*Min*	*Max*
Date registered	10/30/02	433 days	12/16/99	7/22/04
Download	103,973	299,085	1,492	1,985,014
Project activity	3,912	8,370	80	37,297
Target audience	80%	41%		
Software topic	63%	49%		
License	57%	50%		
Total releases	22.8	23.1	4	147
Releases analyzed	6.94	4.13	2	20

4.3. Measures

4.3.1. Modularity. We calculate the *degree of modularity* for each major software release sampled using the source code contained in each release. This measure is based on [15]'s measure of software design quality. Refer to Section 3.2 for details of the calculation.

4.3.2. Software quality. We use measures of intrinsic quality and customer satisfaction to assess software quality. To measure intrinsic software quality, we calculate the *number of static bugs* and *software complexity* based on objective evaluations of the source code included in the software release using two static source code analysis tools.

To assess customer satisfaction, we use three measures of software quality previously identified in OSS research as a proxy for software quality and OSS project success [3]: *number of bugs reported, percentage of bugs closed,* and *time to close bugs*. Refer to Section 3.3 for details of all software quality measures.

[11] Project activity refers to any communication behavior made by OSS project members (e.g. posts to discussion forums, bug trackers, email lists, etc.). Projects with at least 75 total posts to any of the project communication tools (discussion forums, trackers, and email lists) as well as evidence of use of the source code repository were included in the sampling frame.

4.3.3. Control variables. The age of the project and the time between releases likely affects the size of the development team and the quantity of effort that can be contributed for each release. Because each release occurs at a different point in time, two variables are thus used to capture the effect of time. First, *project age* captures the number of weeks between the date that the project was registered on SourceForge and the date that the specified software was released. Second, *development cycle time* is captured by weeks since the previous release. This value is always zero for the first release of a given project. In addition, the flexibility of the OSS license associated with the software has been found to affect user interest in the project and software development effort [17] [23]. Also, characteristics of the type of software being developed and the project community may also influence design and quality metrics, based on the software requirements and the average programming skill level of the project community. We therefore use dummy codes to represent whether a "business-friendly" *OSS license* is used as well as whether the *intended audience* of the software is for computer programmers (e.g. Developers, System Administrators). We also use a dummy code, *software topic*, to represent whether the type of application being developed is for software development or other primarily technical applications (e.g. databases, Internet). Refer to Table 2 for summary means and correlations of all measures.

5. Results

5.1. Overview

Because our dependent variables were significantly skewed, we performed log transformations on all dependent variables for the analyses. Refer to Table 2 for summary means and correlations for degree of modularity, all dependent variables, and control variables. Since our data is a cross-section of OSS projects over time, we use hierarchical linear modeling (HLM) to test the relationships between degree of modularity and the various measures of software quality.

Table 2. Means and correlations for releases analyzed

Measures	Mean	*n*	S.D.	1	2	3	4	5	6	7	8	9	10
1. Modularity	0.58	203	0.20										
2. Complexity	0.82	180	0.32	-0.35									
3. Number static bugs	6.18	180	0.91	0.44	0.11								
4. Number bugs reported	1.07	146	0.98	0.08	0.09	0.24							
5. Percent bugs closed	11%	198	17%	-0.17	-0.03	-0.19	0.36						
6. Time to close bugs	2.66	199	1.81	-0.02	0.29	0.20	0.27	-0.03					
7. Project age	3.65	203	1.67	0.10	-0.03	0.25	0.01	-0.49	0.12				
8. Development cycle	2.22	203	1.51	0.24	-0.02	0.25	0.21	-0.23	0.22	0.63			
9. Intended audience	0.80	203	0.40	0.36	-0.27	-0.07	-0.07	-0.01	-0.17	0.05	0.12		
10. OSS license	0.57	203	0.49	0.31	-0.23	-0.07	0.18	0.05	0.03	0.11	0.17	0.46	
11. Application type	0.76	203	0.42	-0.06	-0.15	-0.30	-0.03	-0.02	-0.09	0.09	0.08	0.24	0.30

We present the results of four models for each quality measure to assess the relationship between degree of modularity and software quality: Model 1: unconditional means (do projects differ in software quality?), Model 2: regression with means-as-outcomes (does modularity explain differences in quality across projects?), Model 3: random coefficient (does modularity explain differences in quality within projects?), and Model 4: the best fitting model (what role does modularity play after accounting for other variables?).[12] Including modularity and all control variables, the full model is represented as follows:

Level-1 model (release effects):

$$\mathrm{Y}_{ij} = \beta_{0j} + \beta_{1j}(\mathrm{Mod}_{ij}) + \beta_{2j}(\mathrm{Age}_{ij}) + \beta_{3j}(\mathrm{Dev}_{ij}) + r_{ij}$$

Level-2 model (project effects):

$$\beta_{0j} = \gamma_{00} + \gamma_{01}(\mathrm{Mod}_j) + \gamma_{02}(\mathrm{Lic}_j) + \gamma_{03}(\mathrm{Aud}_j) + \gamma_{04}(\mathrm{Top}_j) + {}_{0j}$$

$$\beta_{1j} = \gamma_{10} + \gamma_{11}(\mathrm{Mod}_j) + \gamma_{12}(\mathrm{Lic}_j) + \gamma_{13}(\mathrm{Aud}_j) + \gamma_{14}(\mathrm{Top}_j) + {}_{1j}$$

[12] We examined several models using different combinations of modularity, control variables, random error covariance structures, etc., to identify the most appropriate and best-fitting model. We only present the results of the best-fitting model here.

$\beta_{2j} = \gamma_{20} + {}_{2j}$

$\beta_{3j} = \gamma_{30} + {}_{3j}$

where Y_{ij} represents software quality (e.g. software complexty) for release *i* in project *j*, Mod_{ij} is release modularity centered around the project mean, Mod_j is project modularity centered around the grand mean, Age_{ij} is the project age for release *i* in project *j*, Dev_{ij} is the development cycle time for release *i* in project *j*, Lic_j is the license for project *j*, Aud_j is the intended audience for project j, and Top_j is the software topic for project *j*. We used SAS 9.1 (PROC MIXED, GLIMMIX, and GENMOD) to test these models.

5.2. Software complexity

First, we find evidence of significant variation in average software complexity (log) across projects (0.07, s.e. 0.02, *n*=180) (see Model 1, Table 3). Sixty percent of the variation in complexity can be explained by the differences between projects. Second, average project modularity is marginally associated with lower software complexity (-0.46, s.e. 0.26), and explains 8% of project-to-project variation in mean software complexity (Model 2). Third, there is no statistically significant difference in the relationship between modularity and software complexity within projects (-0.33, s.e. 0.27). However, the slopes representing the relationship between modularity and software complexity vary significantly across projects (variance component is 0.69, s.e. 0.38), explaining 70% of the within-project variation in software complexity. The best fitting model using repeated measures (Model 4) includes significant effects for project-level modularity, release-level modularity, and software topic. As degree of modularity increases, level of complexity both across projects (-0.73, s.e. 0.22) and within projects (-0.38, s.e. 0.18) decreases. In addition, projects creating applications associated with software development exhibit software with lower complexity (-0.23, s.e. 0.08). Projects significantly differ in average software complexity (0.06, s.e. 0.02), and projects exhibit significantly different relationships between degree of modularity and software complexity (0.46, s.e. 0.21).

Table 3. HLM results for software complexity

Fixed effects	Model 1	Model 2	Model 3	Model 4
Intercept	0.85***	0.84***	0.85***	1.00***
	0.04	0.04	0.04	0.07
Mod (proj)		-0.46*		-0.73**
		0.26		0.22
Mod (rel)			-0.33	-0.38**
			0.27	0.18
Software				-0.23**
				0.08
Random effects				
Within project	0.04***	0.04***	0.01***	0.02**
	0.01	0.01	0.00	0.01
Initial status	0.07***	0.06***	0.08***	0.06***
	0.02	0.02	0.02	0.02
Rate of change			0.69**	0.46**
			0.38	0.21
Model fit				
-2 LL	26.9	23.9	-113.3	-159.1
AIC	32.9	31.9	-103.3	-143.1
BIC	38.3	39.2	-94.1	-128.5
Intraclass Correlation	0.60	0.59	0.86	0.73

* $p < 0.10$, ** $p < 0.05$, *** $p < 0.0001$

5.3. Number of static bugs

First, the number of static bugs (log) varies across projects (0.68, s.e. 0.16, *n*=180) (see Model 1, Table 4). Eighty-three percent of variation in the number of static bugs can be explained by differences between projects. Second, average project modularity is significantly associated with more bugs (1.65, s.e. 0.78) (Model 2). Average project modularity explains 10% of the between project variation in number of static bugs. Third, similar to software complexity, there is no statistically significant difference in the relationship between modularity and number of static bugs within projects (1.08, s.e. 0.90) (Model 3). However, the relationship between modularity and number of static bugs varies across projects (variance component is 12.31, s.e. 6.58), explaining 44% of the within-project variation in number of static bugs. The best fitting model using repeated measures (Model 4) includes significant effects for project-level modularity, release-level modularity, project age, and OSS license. As degree of modularity increases, the number of static bugs both across projects (2.10, s.e. 0.80) and within projects (1.22, s.e. 0.60) increases. In addition, older projects exhibit more static bugs (0.11, s.e. 0.02), and projects that use a business-friendly OSS license exhibit fewer static bugs (-0.41, s.e. 0.25). Projects exhibit marginally different relationships between degree of modularity and static bugs (5.76, s.e. 3.97).

Table 4. HLM results for number of static bugs

Fixed effects	Model 1	Model 2	Model 3	Model 4
Intercept	6.19***	6.19***	6.19***	6.43***
	0.15	*0.12*	*0.13*	*0.18*

Mod (proj)	1.65***			2.10**
	0.78			*0.80*
Mod (rel)			1.08	1.22*
			0.90	*0.60*
Project age				0.11***
				0.02
Dev cycle				-0.03
				0.02
License				-0.41*
				0.25
Random effects				
Within project	0.14***	0.12***	0.08***	0.72***
	0.02	*0.02*	*0.01*	*0.15*
Initial status	0.68***	0.62***	0.70***	
	0.16	*0.15*	*0.16*	
Rate of change			12.31**	5.76*
			6.58	*3.97*
Model fit				
-2 LL	282.9	277.3	234.1	108.1
AIC	286.9	281.3	240.1	114.1
BIC	290.5	285.0	245.6	119.6
Intraclass Correlation	0.83	0.84	0.90	

* $p < 0.10$, ** $p < 0.05$, *** $p < 0.0001$

5.4. Number of bug reports

We do not find any evidence that projects differ in the number of bugs reported (-0.05, s.e. 0.12, *n*=146).[13] Therefore, none of the variables, including modularity significantly predict number of bug reports since there are no differences to be predicted.

5.5. Percentage of bugs closed

The model predicting percentage of bug reports closed using the full model (see Section 5.1) was statistically significant and the best-fitting model.[14] The predictors release-level modularity, development cycle age, and interactions of release-level modularity with project-level modularity, license, audience, and software topic were each statistically significant. For these data, increasing release-level modularity decreases the percentage of bugs closed (-2.18, s.e. 0.73). Increasing development cycle time increases the percentage of bugs closed (0.11, s.e. 0.05). An increase in one unit of the interaction terms with project-level modularity (18.79, s.e. 6.99) and audience (11.72, s.e. 3.93) increases the percentage of bugs closed. However, an increase in one unit of the interaction terms with license (-9.34, s.e. 2.44) and software topic (-9.05, s.e. 3.62) decreases percentage of bugs closed.

5.6. Time to close bugs

Projects vary significantly in the median time to close bugs (log) across projects (1.55, s.e. 0.47) (see Model 1) (Table 5). Forty-four percent of total variation in time to close bugs occurs between projects. However, degree of modularity does not explain any of this variation either within projects or across projects. Project modularity (Model 2) does not significantly affect time to close bugs (-0.23, s.e. 1.81) (Model 2). In addition, there is no statistically significant difference in the relationship between modularity and time to close bugs across projects (0.01, s.e. 1.37) (Model 3). Between-project modularity and within-project modularity also did not significantly predict time to close bugs in the final model (Model 4).

Table 5. HLM results for time to close bugs

Fixed effects	Model 1	Model 2	Model 3	Model 4
Intercept	2.72***	2.72***	2.72***	3.28***
	0.22	*0.22*	*0.22*	*0.38*
Mod (proj)		0.01		-0.28
		1.37		*1.34*
Mod (rel)			-0.23	0.18
			1.81	*2.00*
Project age				-0.19*
				0.11
Dev cycle				0.27**
				0.11
Topic				-0.83*
				0.46
Random effects				
Within project	1.98***	1.97***	1.83***	1.75***
	0.22	*0.22*	*0.22*	*0.21*
Initial status	1.55***	1.61***	1.60***	1.61***
	0.47	*0.49*	*0.48*	*0.48*
Rate of change			25.19	34.56
			26.51	*29.53*
Model fit				
-2 LL	775.8	773.3	770.7	764.6
AIC	779.8	777.3	776.7	770.6
BIC	783.5	781.0	782.2	776.1
Intraclass Correlation	0.44	0.45	0.47	0.48

* $p < 0.10$, ** $p < 0.05$, *** $p < 0.0001$

[13] Due to the logarithmic distribution of number of bug reports (log), we used PROC GLIMMIX to test this model. This method tests for systematic variation in project means.
[14] Initial analyses suggested the between-project variation is consistent with within-project variation, so we used PROC GENMOD to analyze percentage of bugs closed.

6. Discussion

Contrary to popular belief and theories of modularity, empirically we find a more complex relationship between degree of modularity and software quality. Degree of modularity affects different types of software quality differently. First, we find the expected negative relationship between degree of modularity and software complexity when examining across project differences in average complexity. However, individual projects do not consistently exhibit the same type of relationship between modularity and complexity. As modularity increases, software complexity in some projects decreases substantially, yet increases substantially in others. This suggests that projects do not deal with modularity similarly, so exploration of additional project or release characteristics would help further explain this finding.

Second, we find a positive relationship between degree of modularity and the number of static bugs. Existing theory suggests that increasing modularity should reduce the number of bugs, yet this objective metric suggests otherwise. We do find significant differences in the relationship between modularity an static bugs across projects, similar to software complexity, suggesting that additional project or release characteristics may moderate these relationships. Because the majority of variation in number of bugs occurs between projects, we also need to explore other project attributes that may moderate this relationship to help explain why higher mean modularity is associated with an increase in number of static bugs.

Third, percentage of bugs closed exhibits a very complex relationship with software modularity. As release-level modularity increases, the percentage of bugs closed unexpectedly decreases. However, projects with higher mean modularity and increasing modularity within the projects close a higher percentage of bug reports. In addition, projects with software that targets software developers and increasing modularity over releases also close a higher percentage of bug reports. However, projects that use a business-friendly license or create applications for software development and exhibit increasing modularity across releases close a lower percentage of bugs. Finally, we do not find any empirical support for a relationship between software modularity and number of bug reports, or time to close bugs.

7. Limitations

While we introduce new measures of modularity and software quality and test our hypothesis using longitudinal data from a fairly large sample of OSS projects, our study is not without limitations. While we are able to control for differences in software quality due to differences in tools used by sampling projects hosted only on SourceForge, we do not consider projects hosted on other sites. In addition, we examine projects written only in the Java programming language. While we do not anticipate reasons our results may not be generalizable to other object-oriented programming languages, this should be explore empirically. Other types of structure and project characteristics may also help to explain the unexpected negative relationship between degree of modularity and software defects, such as project governance mechanisms and the skill level of code contributors. We hope to further explore these variables in future research using our data. Finally, while we used several measures to assess software quality, we lack corroborating evidence from code contributors to support our results. In future research, we hope to address these concerns by collecting survey and interview data from code contributors and other project members to address perceptions about software quality as well as their experience writing software for the project.

8. Conclusion and Implications

We empirically examine the relationship between software modularity and software quality in this paper. By introducing a new measure of modularity, we assess the relationships among software components as packages, not classes or files, which is a more pragmatic assessment of software modularity. We also assess software quality in OSS projects using two new measures: software complexity and number of static bugs. In addition, we apply the previously used bug report measures to assess software quality, yet we adjust these measures to account for the size of the user base.

Our empirical findings both complement and challenge existing theories of modularity. While existing theory suggests that degree of modularity should improve software quality, the data suggests this relationship is quite complex. As expected, we find a decrease in software complexity as degree of modularity increases. However, we see an increase in number of software defects as modularity increases. In both cases, modularity does not exhibit consistent relationships with these quality measures across projects. Data also suggest a complex relationship between modularity and percentage of bugs closed, with several control variables moderating this relationship.

The empirical findings regarding the relationship between modularity and product quality have implications for the practice of software development. These findings may prove useful to software development project managers who need practical and quantifiable techniques for monitoring the software development process and subsequent software quality. The measures of modularity and software quality empirically examined and tested may also help software development managers evaluate existing software and software under development.

References

[1] Baldwin, C. Y. and Clark, K. B. "Managing in an age of modularity." Harvard Business Review, 75, 84-93. (1997).

[2] Baldwin, C. Y., & Clark, K. B. Design rules: The power of modularity (Vol. 1). Cambridge: The MIT Press. (2000).

[3] Crowston, K., Annabi, H., Howison, J., & Masango, C. Towards a portfolio of FLOSS project success measures. Paper presented at the Collaboration, Conflict and Control: The 4th Workshop on Open Source Software Engineering, International Conference on Software Engineering, Edinburgh, Scotland. (2004, May 25).

[4] Flanagan, D. Java in a Nutshell (3rd Ed.). Sebastopol: O'Reilly & Associates, Inc. (1999).

[5] Glover, A. "In pursuit of code quality: Monitoring cyclomatic complexity," Retrieved 3/5/2007, from http://www-128.ibm.com/developerworks/java/library/j-cq03316/ (2006a).

[6] Glover, A. "In pursuit of code quality: Tame the chatterbox." Retrieved 3/5/2007, from http://www-128.ibm.com/developerworks/java/library/j-cq06306/ (2006b).

[7] Glover, A. "In pursuit of code quality: Code quality for software architects." Retrieved 5/15/2008, from http://www.ibm.com/developerworks/java/library/j-cq04256/ (2006c).

[8] Gumpta, S. "Package design." Retrieved 2/22/2008, from http://javaboutique.internet.com/tutorials/PackageDesign/index.html (2008).

[9] Johnson, J. P. "Open source software: Private provision of a public good," Journal of Economic and Management Strategy, 11(4), 637-662. (2002).

[10] Kan, S. Metrics and Models in Software Quality Engineering (2nd Ed). Reading: Addison Wesley Professional. (2002).

[11] Lincke, R. "Validation of a Standard- and Metric-Based Software Quality Model – Creating the Prerequisites for Experimentation." Licentiate Thesis. Reports from MSI. Växjö University. (2007).

[12] Liu, X., & Iyer, B. "Design architecture, developer networks, and performance of Open Source Software projects," Proceedings of the Twenty-Eighth International Conference on Information Systems (ICIS2007), Montreal, Canada. (2007).

[13] MacCormack, A., Rusnak, J., & Baldwin, C. "Exploring the structure of complex software designs: An empirical study of open source and proprietary code," Management Science, 52(7), 1015-1030. (2006).

[14] MacCormack, A., Rusnak, J., & Baldwin, C. "The impact of component modularity on design evolution: Evidence from the software industry," Harvard Business School Working Paper. (2007).

[15] Martin, R. "OO design quality metrics: An analysis of dependencies," Retrieved 1/13/2006, from http://www.objectmentor.com/resources/articles/oodmetrc.pdf (1994).

[16] McCabe, T. J. "A complexity measure," IEEE Transactions on Software Engineering, 2(4), 308-320. (1976).

[17] O'Mahony, S. Guarding the commons: How community managed software projects protect their work. Research Policy, 32, 1179-1198. (2003).

[18] Parnas, D. L. "On the criteria to be used in decomposing systems into modules," Communications of the ACM, 15(9), 1053-1058. 1972.

[19] Parnas, D. L., Clements, P. C., & Weiss, D. M. "The modular structure of complex systems," IEEE Transactions on Software Engineering, 11(3), 259-266. (1985).

[20] Sanchez, R., & Mahoney, J. T. "Modularity, flexibility, and knowledge management in product and organization design," Strategic Management Journal, 17, 63-76. (1996).

[21] Simon, H. A. (1969). Sciences of the Artificial. Cambridge, MA: MIT Press.

[22] Software Productivity Consortium, "Software Measurement Guidebook", International Thomson Computer Press. (1995).

[23] Stewart, K. J., Ammeter, T. A., & Maruping, L. "Impacts of license choice and organizational sponsorship on user interest and development activity in open source software projects," Information Systems Research, 17(2), 126-144. (2006).

[24] Sullivan, K. J., Griswold, W. G., Cai, Y. & Hallen, B. "The structure and value of modularity in software design," In the Proceedings of the 8th European Software Engineering Conference, Vienna, Austria. (2001).

[25] Ulrich, K. T. "Role of product architecture in the manufacturing firm," Research Policy, 24, 419-440. (1995).

[26] W3Schools. "Browser Statistics." Retrieved 2/21/2008, http://www.w3schools.com/browsers/browsers_stats.asp (2008).

[27] Wheeler, D. A. "Why Open Source Software/Free Software (OSS, FLOSS, or FOSS)? Look at the numbers!" Retrieved 2/21/2008 from http://www.dwheeler.com/oss_fs_why.html (2007).

[28] Witt, B., Baker, T., & Merritt, E. Software architecture and design. New York: Van Nostrand Reinhold. (1994).

[29] Zhu, H. Software design methodology: From principles to architectural styles. Oxford: Elsevier. (2005).