The Impact of API Complexity on Failures: An Empirical Analysis of Proprietary and Open Source Software Systems

Marcelo Cataldo[1], Cleidson R.B. de Souza[2]

June 2011
CMU-ISR-11-106

Institute for Software Research
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3890

[1] Institute for Software Research, School of Computer Science, Carnegie Mellon University
[2] IBM Research, Brazil

ABSTRACT

Information hiding is a cornerstone principle of modern software engineering. Interfaces, or APIs, are central to realizing the benefits of information hiding, but despite their widespread use, designing good interfaces is not a trivial activity. Particular design choices can have a significant detrimental effect on quality or development productivity. In this paper, we examined the impact of API complexity on the failure proneness of source code files using data from two large-scale systems from two distinct software companies and nine open source projects from the GNOME community. Our analyses showed that increases in the complexity of APIs are associated with increases in the failure proneness of source code files. Interestingly, there are significant differences between corporate and open source software. Although the impact of the complexity of APIs is important in both settings, the magnitude of the detrimental effects on quality is significantly higher in corporate settings. We discuss the research and practical implication of the results.

Keywords: API complexity, corporate versus open source, software failures.

1. INTRODUCTION

The concept of information hiding proposed by Parnas [32] is a fundamental principle that allows software architects, designers and engineers to develop modular software systems. This concept has been instantiated as data encapsulation, interfaces, polymorphism and other ways in modern programming languages [27]. Interfaces or APIs, in particular, are widely regarded as "the only scalable way to build systems from semi-independent components" [20]. They allow software developers to work in parallel and minimize the impact of their colleagues' work [12, 32]. As we witness the rapid development of new software platforms and ecosystems (e.g. mobile platforms), it is almost trivial to recognize the growing importance that APIs' have, particularly in terms of its use and acceptance in the industry. Many companies today such as Apple, Google, and Microsoft, just to name a few, rely on APIs to leverage part of their businesses.

Despite APIs' importance in the industry [13, 14], designing good APIs is not a trivial activity [16, 24]. While there are different guidelines for an API implementation [6, 14], they are solely based on experience and anecdotal evidence collected by the authors of the guidelines. In addition, past work has examined how design choices in terms of API design impact quality attributes such as maintainability [2] and usability [16]. Although one could argue that elements associated with the complexity of APIs have been addressed by the research on API usability, limited attention has been given to systematically evaluating the impact of design attributes of APIs such as complexity on traditional outcomes such as software quality and development productivity. This lack of research on the relationship between "good" API design and its impact on software projects is a gap that needs to be addressed because it represents a potential barrier for development organizations towards fully realizing the benefits of modular systems.

Furthermore, the modularity literature (e.g. [36]) argues that APIs are the link between modules or components that allows for separation of concerns (a technical aspect) and development activities, therefore facilitating the coordination among developers (an organizational aspect). Such argument rests on the implicit assumption that complexity is embedded in the software entity (e.g. module or component) and not in the APIs themselves. Recent research suggests a departure from that line of thought because it suggests that complexity might in fact be embedded in the APIs themselves (e.g. [2, 12, 35]). As a consequence, it is critical to understand whether APIs have the potential to become barriers to effectively decoupling technical responsibilities and work responsibilities as well as hinder coordination among development teams.

In this paper, we examine the relative impact of API complexity on the failure proneness of source code files using data from two large-scale systems from two distinct software companies as well as nine projects from the GNOME community. We assessed API complexity using measures proposed by Bandi and colleagues [2],.Our empirical analyses showed that increases in the complexity of APIs are associated with increases in the likelihood of source code files being part of a post-release defect while controlling for several other factors that lead to software failures. More interestingly, the negative impact of API complexity is more acute in the context of corporate projects than in open source projects. Combined, these results suggest, first, that attention should be given to the design of APIs given their important role in terms of software quality and, second, that further research should focus on understanding the underlying factors that render open source projects less vulnerable to the negative impact of API complexity.

The rest of the document is organized as follows. First, we discuss previous research that motivates the research questions examined in this paper. Second, we present our empirical analyses involving 11 software projects, two from corporate settings and 9 from the GNOME open source community. We conclude with a discussion of the limitations and implications for future research of our work.

2. THE ROLE OF API COMPLEXITY ON SOFTWARE FAILURES

Design decisions have an important impact on the ability of software systems to achieve their functional and non-functional requirements [5, 37]. Principal decisions articulated at the architectural level serve as a general framework that guide and constrain lower level and more detailed design decisions [37]. Two critical and interrelated attributes of a software system are impacted by design decisions: the allocation of functional responsibilities within specific constituent parts of a system (e.g. modules or components) and relationships among those parts.

Those relationships are typically realized in the form of APIs, which play a very important role in the development process of any software system from a technical point of view as well as from an organizational point of view.

In the technical dimension, APIs determine a host of attributes of a system such as the efficiency of the communication between modules or components (e.g. [37]), the ease and efficiency of accessing the functionality of a module or a component (e.g. [25]), the usability and understanding of the APIs (e.g. [16]), as well as the evolution and maintainability of the system (e.g. [24]). Poorly designed APIs could represent an important technical liability for software systems, particularly, for those large-scale complex systems that are pervasive in today's world.

On the organizational side, the design of a module or a component API defines a key set of coordination needs for a development organization. Unsatisfied coordination needs tend to results in misunderstandings and mistakes, which typically manifest themselves as higher levels of defects (e.g. [10, 35]). Moreover, uncertainty about the attributes of an API (e.g. number and type of parameters) creates a number of coordination requirements that tend to be difficult to identify [20, 35].

Designing good APIs in technical and organizational terms is, unfortunately, not a trivial task [24]. Despite the fact that guidelines have been proposed for that [6, 13], they are based on the anecdotal evidence and experience by a subset of practitioners. Furthermore, to the best of our knowledge, there is no systematic empirical evaluation of the proposed guidelines. More importantly, only a limited set of studies has investigated how the design of APIs relates to quality attributes of a system like usability and maintainability (e.g. [16]). However, maintainability and usability are two dimensions of a larger set of dimensions about a software system, including complexity, reusability, evolution, and modifiability.

An aspect of API design that has been mostly neglected by researchers is design complexity. One traditional view of complexity focuses on properties of the implementation of a particular function or methods and an extensive literature has developed since Halstead's and McCabe's seminal work on code complexity [22, 28]. Another influential view of design complexity relates to structural properties of a software system and the interconnections among constituent parts. The concepts of coupling and cohesion are central in this line of work. The dimension of design complexity in the context of APIs represents a third line of work that has received limited attention: it focuses on the complexity of the interconnections themselves, instead of the number of interconnections among the parts. In particular, to the best o our knowledge, there is limited work focusing on the empirical analysis of API complexity measures and their impact on traditional software engineering outcome variables such as quality and development productivity. One such example is the work of Bandi and colleagues [2] who examined the impact of design complexity of APIs in maintenance tasks of software systems. The authors used a set of metrics (e.g. API size and operation argument complexity) to assess the complexity of APIs. These metrics are based on the types and the number of parameters a method or a function has. They are somewhat intuitive: they rely on the assumption that a method or function with a large number of parameters and whose parameters are objects is said to be more complex than another method or function with fewer parameters based on primitive types (e.g. integers). Bandi and colleagues [2] found that maintenance tasks with higher levels of API complexity took longer to resolve than those that involved less complex APIs. In addition, Bandi's results indicate that their metrics are redundant (i.e., measuring similar properties of the system design) and therefore suggests that the usage of only one metric is enough. Their work provides empirical evidence that API complexity is detrimental to development productivity during the maintenance of software systems. On the other hand, the relationship of API complexity with software quality has been neglected. This paper addresses this gap in the literature by examining the following research question:

RQ1: ***What is the impact of API complexity on failure proneness of software systems***?

There has been recent work about the modularity of open source vs. proprietary systems. While some research has argued that open source software systems tend to be more modular than proprietary systems [27], other work has not found evidences supporting such argument (e.g. [33]). If open source is in fact more modular, then, open source development, in principle, can better realize the benefits of modularization such as reduced maintenance, testing and experimentation costs [36]. Since the design of APIs is an important step in the design of modular systems, we are also interested in assessing the impact of API complexity on failure proneness in the context of open source projects. More specifically, we address the following research question:

RQ2: ***Does the impact of API complexity on failure proneness differ in proprietary systems relative to open source systems***?

3. EMPIRICAL METHODOLOGY

In this section, we describe the empirical design used in our inquiry on the impact of API complexity on software failures. The section is organized as follows. We first describe the characteristics of the projects from which data were collected. Second, we describe the various measures used in our analyses followed by a discussion of the statistical modeling used in the study.

3.1 Description of the Projects

Our dataset comprise of data from two large software development projects from two distinct companies and nine projects from the GNOME open source community.

3.1.1 Corporate Projects

One of the corporate projects was a complex distributed system produced by a company operating in the computer storage industry. One hundred and fourteen developers grouped into eight development teams distributed across three development locations, worked full time on the project during the time period covered by our data. The data covered a period of 11 months of development activities corresponding to the latest release of the company's product. Those development activities were captured by 1125 development tasks, which involved activities from implementation of new features to fixing defects. The system was composed of approximately 5 million lines of code with most of the code written in C and a relatively small fraction in C++. The API information was collected using the C-REX tool.

The second was an embedded system from a company in the automotive industry. This development organization used a product line approach. Our data covered 4 years of development activities in the latest version of the base platform software grouped in 3,840 development tasks. Three hundred and eighty developers distributed in eight locations across Europe and Asia participated in the development. The system was composed of approximately 7 million lines of code organized in 530 architectural components. All source code files were written in C language. The API-related information was collected using a proprietary tool that the development organization used to collect a wide range of metrics for each source code file.

3.1.2 Open Source Projects

We also collected data from nine open source projects from the GNOME community. This community has developed a graphical user interface platform for operating systems such as Linux. It was initiated in 1997 and over the following decade, volunteers around the world have contributed to this development effort in order to create a freely available desktop platform and a host of applications. GNOME is in fact a compilation of various software development projects [18]. Today the community has over 730 projects. There are only few, relatively easy to meet, requirements to create a new project in the GNOME source code repository system. Therefore, projects differ significantly in their development activity, size, and participation rate. Building on criteria used in past research [11], we only considered projects that satisfied the following criteria: (a) continuity of development activity (at least one year), (b) amount of development activity (at least 200 commits), (c) attractiveness of project for developers (at least 10 committers), and (d) user interest (at least one community hosted mailing list). Additionally, data from the mailing lists and data collected from the source code repository should overlap during the analyzed period. These criteria were met by 90 projects. These 90 projects range from large and long-standing projects such as Evolution to smaller and more recent projects such as Cheese. Unfortunately, these 90 projects represent a major amount of data to be processed in order to measure the relevant factors in our analyses. Then, we opted for selected a random sample consisting of 9 projects (a 10% sample). The resulting sample included the following 9 projects: balsa, brasero, evolution, gnome-color-manager, memprof, orbit2, planner, seahorse, and vala. The size of the projects in terms of non-comment non-empty lines of code ranged from 11K to 329K. These projects used several different programming languages including C, C++, perl, and python. Therefore, we used the tool Doxygen to extract the API information from the various source code files in a consistent manner.

The dataset covered almost the last 13 years of activities in the community, from November 1997 until October 2010. During that time period, the projects in our random sample projects had several releases. However, we focused our analyses on the latest release of each of the projects.

3.2 Description of the Measures

The basis of our data collection was the development activity represented by changes in the source code. In both corporate projects, every change to the source code was controlled by modification requests. A modification request (MR) represents a development task such as implementing a new functionality or resolving a defect. In short, the version control system and the MR-tracking systems of the development organizations constituted the main sources of data. In the open source projects, the link between modification requests (or reports in the GNOME Bugzilla's database) and commits in the version control is not consistent. Then, we focused mostly on the version control data. In addition, we utilized the source code itself as a third important source of data to extract additional information about technical properties of the systems. Combining all three data sources, we constructed the measures described in the subsequent paragraphs.

3.2.1 Measuring Software Failures

Our measure of quality is failure proneness defined as the likelihood of a particular software entity (e.g. source code file, module, component, etc) to be modified as part of fixing a "field" defect. Our unit of analysis is the source code file. Then, the dependent variable, *File Buggyness*, was measured as a dichotomous variable indicating whether a source code file has been modified in the course of resolving a "field" defect. In the case of the corporate project, the strong connection between modification requests tracking processing and version control system allowed us to relatively easily identify the defects that were associated with each particular source code files. In our first corporate project, field defects represent instances of problems reported by customers after the product has been released. In the case of second project (the embedded automotive system), we consider "field" defects as those encountered in the integration and system-testing phase of the development process. Field defects in that industry seldom occur because they would result in product recall and significant financial impact. Then, an important amount of effort is spent in the system-testing phase to make sure that defects are found before the product is released to customers.

In the case of the open source projects, we considered field defects as those reported in the GNOME's Bugzilla database against the last release of each project. One challenge in these projects is the lack of a reliable way to link the commits that fixed the defects with the corresponding defect report in Bugzilla. Then, we manually inspected the identified defects for all 9 projects. Some of the defect reports contained information about the associated commits in the form of comments or the commits messages contained references to defect report. In those cases (37% of the total number of identified defects), we use this information to compute our outcome measure. For those defects that did not contained information about the associated commits, we inspected the version control data and we looked at commits submitted around the time the defect was marked as resolved. Specifically, we looked at the commits that occurred within the +/-4 hours time window of resolving the defect. Two raters unaware of the research questions posed in this paper examined the comments in the commits and in the defect reports and connected them if they found a relationship between the problem description (in the defect report) and the explanation of the resolution (in the commit). Then, this information was also used to compute our *File Buggyness* measure for the open source projects.

3.2.2 Measuring API Complexity

As discussed before, Bandi and colleagues' [2] results indicate that instead of using a set of metrics he proposed, a single metric could be use, because all metrics were measuring similar properties. Following this rationale, we constructed several measures of API complexity based on the metrics proposed by Bandi and colleagues. For each source code file, we first identified the set of APIs those files made accessible for other source code files (e.g. public methods and variables for C++ code and *extern*-ed functions and variables for the C portion of the code). Then for each API, we computed the interface or API size as described by Bandi and colleagues [2]. API size is defined as the number of parameter plus the sum of the parameters' type sizes and both terms are multiplied by constants. In our computations, we set those constants to 1 (see [2] for a discussion of selecting the constant values). Bandi and colleagues [2] did not consider the case where APIs are data elements such as global variables. In those cases, we can define the API size measure in different ways. One possibility is to assume that it is equivalent to a "parameterless" API and assign a 0 to this measure. Alternatively, we could assign to the API size measure the value corresponding to the data element type as if it were a regular API with a single parameter. We evaluated both approaches and the results were similar. We report the results based on the second approach. Finally, the API-level measures were aggregated at the level of the source code file. For each file, we calculated a total sum measure of API size and its standard deviation. Then, we have two variables: *API Size,* and *API Size Dispersion*.

3.2.3 Additional Factors Impacting Software Failures

In order to adequately examine the impact API complexity has on software failures in proprietary and open-source systems, we need to account for the effects of potentially confounding influences, our analysis must include factors that past research has found to be associated with failures. When it comes to prediction of software failures, numerous measures have been evaluated in corporate as well as open source settings (e.g. [15, 19, 21, 29, 30, 40]). As suggested by Graves and colleagues [19], such measures can be classified as either process or product measures. Process measures such as number of changes or deltas, and age of the code (i.e., churn metrics) have been shown to be very good predictors of failures [19, 29]. Accordingly, we control for the *Number of Commits*, which is the number of times the file was changed as part of some development activity prior to the project's release. We also control for the *Average Number of Lines Changed* in a file as part of the development activities as well as for the *Number of Developers* that modified the file during the project. In contrast, product measures such as code size and complexity measures have produced somewhat contradictory results as predictors of software failures. Some researchers have found a positive relationship between lines of code and failures (e.g. [8]), while others have found a negative relationship (e.g. [4]). We measure *Size of the File (LOC)* as the number of non-blank non-comment lines of code.

We also collected measures of experience based on the approaches used by Boh and colleagues [7] which utilize the data in software repositories as the basis for assessing experience. We measured the experience of the developers that modified file *i* prior to the release of the product we studied, *Average Experience of Developers,* as the average number of commits made by the developers that worked on modification to file *i*. We were able to measure this particular factor because our data covered also development activities in all 11 projects prior to the work associated with the last release of the project.

Recent research has shown that how software entities such as components, modules and source code files are inter-related through their technical dependencies, syntactic or logical, impacts the quality of those software entities (e.g. [10 , 29, 40]). Syntactic dependencies were extracted from the various systems in different ways. We used the C-REX tool [23] to identify programming language tokens and references in each entity of each source code file of the distributed system (first corporate project). In the second corporate project, the embedded system, the syntactic dependencies of each source code file were extracted by a proprietary tool, which the development organization used to collect several metrics for each file. Finally, the syntactic dependencies in the nine open source projects were collected using Doxygen's cross-reference generation feature. Using the collected data about syntactic dependencies, we merged the information into a matrix for each project that captures the relationships amount the system's constituent source code files. Those relationships identified data, function and method references that crossed the boundary of each source code file. More precisely, in a file-to-file matrix SD, the cell sd_{ij} represents the number of data/function/method references that exist from file *i* to file *j*. We refer to data references as *data dependencies* and function/method references as *functional dependencies*. We collected four syntactic dependencies measures: inflow and outflow data relationships and inflow and outflow functional dependencies. Each of those four measures capture the number of syntactic dependencies of such type associated with each file *i*.

Logical dependencies are another type of technical relationship that might exist among software entities. Logical dependencies relate source code files that are modified together as part of a unit of development work. When a development task requires changes to more than one file, we assume that decisions about the change to one file depend in some way on the decisions made about changes to the other files involved in the MR. This view of logical dependencies been empirically assessed in a number of corporate (e.g. [9, 10]) and open source projects (e.g. [38]). In the corporate projects, the modification requests contained information about the commits made in the version control system. As described earlier, such information was reliably generated as part of the submission procedures established in the development organizations. Such data allowed us to identify the relationship between development tasks and the changes in the source code associated with such tasks. Using this information, we constructed a logical dependency matrix. In the case of the open source projects, we considered each commit as a unit of work and, therefore, it was used to construct the logical dependency matrix for GNOME projects we studied. More specifically, the logical dependency matrix is a symmetric matrix of source code files where each cell C_{ij} represents the number of times files *i* and *j* (*i* is not equal to *j*) were changed together as part of a development task[1]. We accumulate the data across all the development activities performed as part of the release of each project that we studied. Cataldo and colleagues [10] found that two measures based on logical dependencies – *Number of Logical Dependencies* and *Clustering of Logical Dependencies* – have a major impact on software failures. Building on those results, we computed the corresponding measures as follows. The *Number of Logical Dependencies* measure for file *i* was computed as the number of non-zero cells on column *i* of the matrix. Since the logical dependencies matrix is symmetric, this measure is equivalent to the degree of a node in undirected graph, excluding self-loops. The *Clustering of Logical Dependencies* measure captures the degree to which the files that have logical dependencies to the focal file *i* have logical interdependencies among themselves. Formally, and in graph theoretic terms, the *Clustering of Logical Dependencies* measure for file *i* is computed as the density of connections among the direct neighbors of file *i*. This measure is equivalent to Watts's [39] local clustering measure.

The quality of the logical dependency data, and consequently the measures based on them, relies on the adherence of developers' actions to the defined change submission processes. For instance, a developer could submit a commit containing changes to two different files but those changes are associated with different modification requests and they do not related to an actual dependency among the files. A collection of analyses was performed to assess the quality of our MR-related data in the corporate projects and commit data in the open source projects in order to minimize measurement error. In the case of the corporate projects, we compared the revisions of the changes associated with the modification requests and we did not find evidence of such type of behavior. One of the authors together with a senior engineer of each organization examined a random sample of modification requests to determine if developers have work patterns that could impact the quality of our data such as the example described above. We did not find commits in the version control systems that contained modifications to the systems' code that was unrelated to the development task represented by the modification requests. In the case of the open source projects, two raters unaware of the research questions examine random samples of commits from all nine projects. Although they did not found strong evidence that commits contained unrelated changes to the source code, they did found certain change patterns that might impact the analyses. These changes involved modifications of two different types: (1) changes to configuration files and (2) changes to include files that contained an unusual large number of definitions. We identified all these particular files and removed them from the construction of the logical dependency matrix. Since case (2) could be related to design decisions made in the project, we ran our analyses with and without considering those files and the results were consistent, so the results reported in the paper are based on the analysis that did not included files in case (1) nor in case (2).

Finally, we included a few additional controls. First, we included a binary variable, *Corporate Project*, that was set to 1 if the file I was part of one of the two corporate projects, otherwise, it was set to 0. Second, we also included dummy variables to control for unobserved, project-specific factors. These dummy variables effectively control for any factors, which we could not explicitly measure such as the cultural or organization aspects of the projects.

3.3 Description of the Statistical Model

Our dependent measure, *File Buggyness*, is a binary variable. Then, our analyses used logistic regression models to assess the impact of API complexity on failure proneness. We report two goodness-of-fit measures for each statistical model including the log-likelihood of the model and the percentage of deviance explained by the model. Deviance is defined as -2 times the log-likelihood of the model. The percentage of the deviance explained is a ratio of the deviance of the null model (containing only the intercept), and the deviance of the final model.

Regression coefficients are typically reported as part of the results. We opted for reporting the odds ratios associated with the logistic regressions because they simplify the interpretation of the results. Odds ratios are the expo-

[1] The diagonal of the matrix indicates the number of times a single file was modified and can be disregarded from further analysis.

nent of the logistic regression coefficient. An odds ratio that is larger than 1 indicates a positive relationship between the independent and dependent variables. On the other hand, an odds ratio less than 1 indicates a negative relationship. For example, if the binary measure *Corporate Project* has an odds ratio of 4.4, increasing the value of such factors from 0 to 1 results in an increase of the probability of a file having a reported defect by 4.4 times while the remaining factors in the model are kept constant. We used the *logistic* command in the statistical package Stata 11 for computing the regression models.

4. RESULTS

4.1 Preliminary Analysis

The first step in our analyses consisted in understanding the general characteristics of the data. We calculated descriptive statistics of all the measures and they showed that most of the variables were highly skewed. Therefore, we log-transformed them. We also performed collinearity diagnostics on the measures. A pair-wise correlation analysis reveal several high levels of correlation between our variables, in particular, among Number of Commits, Number of Developers, Size of the File and the syntactic dependency measures. In order to further examine these high level correlations, we compute variance inflation factors (VIF) and tolerances for all the independent and control variables described in section 3.2. A tolerance close to 1 indicates little multicollinearity, whereas a value close to 0 suggests that multicollinearity may be a significant threat. VIF is defined as the reciprocal of the tolerance.

In Table 1, we report the VIFs and tolerances associated with our variables. Traditional rule of thumb suggest that VIF values above 10 should be avoided and values above 5 should be consider with care [26]. We observe in model I in table 1 that several of our variables have VIFs significantly higher than 10. Therefore, we remove those factors from our analyses. Model II (in table 1) shows that the remaining variables have VIFs values below 5 and those factors were the only one considered in our regression analyses. The pair-wise correlations among the variables included in the regression models were sufficiently low to not represent a concern in terms of multicollinearity.

Table 1: Collinearity Diagnostics

	Model I	**Model II**
	VIF (Tolerance)	**VIF (Tolerance)**
API Size	16.39 (0.0617)	1.19 (0.8394)
API Size Dispersion	5.09 (0.2007)	1.07 (0.9334)
Number of Commits	7.75 (0.1297)	---
Average Change Size	6.26 (0.1595)	1.19 (0.8433)
Number of Developers	5.34 (0.1876)	1.43 (0.6972)
Size in LOCs	1.35 (0.7301)	1.29 (0.7780)
Average Experience of Developers	1.09 (0.9205)	1.08 (0.9228)
Syntactic In-Data Dependencies	5.19 (0.1899)	---
Syntactic Out-Data Dependencies	2.89 (0.3527)	1.28 (0.7794)
Syntactic In-Functional Dependencies	1.37 (0.7250)	1.10 (0.9122)
Syntactic Out-Functional Dependencies	1.12 (0.9003)	1.11 (0.9041)
Logical Dependencies	3.27 (0.3001)	1.27 (0.7863)
Clustering of Logical Dependencies	1.27 (0.7859)	1.25 (0.8000)
Corporate Project	4.62 (0.2149)	4.62 (0.2163)
Project Dummy A	1.37 (0.7359)	1.36 (0.7380)
Project Dummy B	2.59 (0.3849)	2.59 (0.3855)
Project Dummy C	3.08 (0.3235)	3.08 (0.3249)
Project Dummy D	3.23 (0.3102)	3.23 (0.3097)

Table 2: The Impact of API Complexity of Software Quality

	Model I	Model II	Model III
Average Change Size	1.008	1.030	1.025
Size in LOCs	1.345**	1.358**	1.323**
Number of Developers	6.564**	6.592**	6.563**
Average Experience of Developers	0.648**	0.689**	0.711**
Syntactic Out-Data Dependencies	1.050**	1.079**	1.040**
Syntactic In-Functional Dependencies	0.997	0.953	1.001
Syntactic Out-Functional Dependencies	1.075*	1.059*	1.109*
Logical Dependencies	1.951**	1.939**	1.937**
Clustering of Logical Dependencies	0.523**	0.517**	0.521**
Project Dummy A	0.753**	0.751**	0.761**
Project Dummy B	1.277**	1.275**	1.279**
Project Dummy C	0.347**	0.379**	0.382**
Project Dummy D	0.309**	0.301**	0.304**
Corporate Project	4.426**	4.552**	4.625**
API Size		1.512**	1.506**
API Size Dispersion		0.931	0.933
API Size X API Size			0.992
API Size X Corporate Project			1.193**
Log-Likelihood	-11718.4	-10096.5	-9578.9
Deviance Explained	43.97%	51.72%	54.19%

(+ $p < 0.10$, * $p < 0.05$, ** $p < 0.01$)

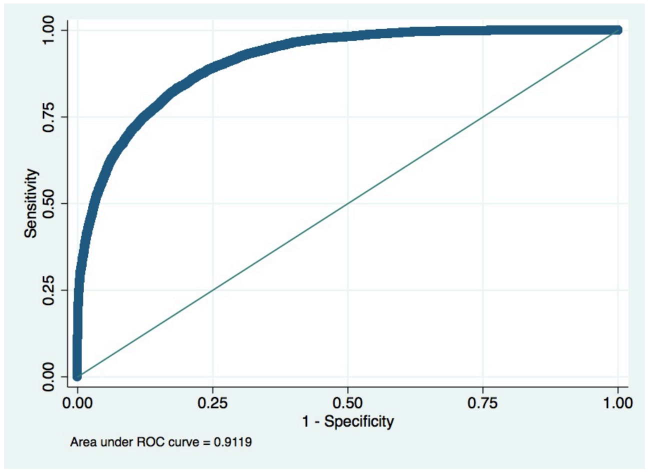
4.2 The Impact of API Complexity on Software Failures

Table 2 reports the results of examining of the impact of API complexity on failure proneness. We report the odds ratios associated with three different regression models. Model I is a baseline model and includes all the control measures that did not exhibit multicollinearity problems. Overall, the results are consistent with past research (e.g. [8, 10, 40]). Factors such as size of the source code file, number of developers that modified a file, number of technical dependencies (syntactic and logical) have odds ratios above 1 indicating that higher levels of those measures increase the likelihood of failures being associated with the source code file. In addition and also consistent with past research (e.g. [10]), we observe that developers' experience and the structural properties of the logical dependencies decrease the likelihood of failures. The dummy variables that controlled for unobserved and unmeasured aspects of the projects are all statistically significant indicating that there are important differences across all 11 projects. Finally, the most important result reported in model I is the statistical significance of the *Corporate Project* variable that indicated whether the focal file was part of a corporate project (a value of 1) or part of an open source project (a value of 0). An odds ratio of 4.426 associated with the factor indicates that source code files in corporate projects are 4.4 times more likely to have failures than files in open source projects. It is important to reiterate that we are controlling for a number of other factors that lead to software failures. Then, this particular result suggests the existence of aspects inherent to the two corporate projects we studied (and arguably to corporate projects in general) that makes achieving levels of software quality as high as in open source projects more challenging. Certainly one key difference between our corporate projects and the open source projects is the size in lines of code. The corporate projects are an order of magnitude larger than the open source project. Although problems stemming from the shear size of the systems might be an important driver of this result, other factors such as the pace of the work, the role of milestones and compressed cycle times made be at play.

Model II introduces our two independent measures, *API Size and API Size Dispersion*. We observe that the impact of API Size is statistically significant. Furthermore, the odds ratio is about 1 indicating that increases in the complexity of the APIs exported by a source code file increase the likelihood of source code files of being associated with defect by 51.2%. While the sum of API sizes gives a general indication of the complexity associated with interacting with a particular file, the dispersion of the API size measure provides an indication of how such complexity is distributed across the APIs exported by a particular file. Model II shows, however, that the impact of dispersion of API sizes is not statistically significant across all projects. It also important to highlight that the effect size of the API Size factor is quite important accounting for almost 8% of the deviance explained in the model. Finally, the quality of the statistical models as evidenced by the respectable levels of deviance explained is strengthen the ROC curve analysis depicted in figure 1 which indicates an area under the ROC curve of 0.9119. Combined, these values are indicative of very good fit of the models.

The last model in table 2, model III, includes two interaction terms. These terms allow us to examine how the impact of a particular factor on the outcome variable changes as a second factor varies. For example, the interaction term *API Size X Corporate Project* tests whether the impact of the API Size variable differs for corporate versus open source projects. The second interaction term, *API Size X API Size*, allows us to examine whether the impact of API Size is purely linear or it has a quadratic dimension to it. The results reported in model III show that only one of the interaction terms in statistically significant. The odds ratio above 1 associated with the term *API Size X Corporate Project* indicates that the impact of API complexity is more acute in corporate projects than in open source projects. In other words, above and beyond the direct effect of API Size, the interaction term tells us that the impact of API complexity is even greater in corporate projects. We explore these results further in the next section.

Figure 1: ROC Curve for Model II from Table 2



4.3 Additional Analyses

We performed additional analyses to examine the robustness of our results. First, we performed regression analyses equivalent to those reported in Table 2 for each project. In this way, we can assess whether the regression models are stable across the various projects. The results in terms of the statistically significant factors, the relative magnitude of the effects as well as the fit of the models were similar to those reported in Table 2. Since our analysis focused only on the last release of each project, we also performed our analyses in previous releases for those projects were had all the relevant data available (all projects except the embedded system corporate project). We ran individual project regressions as well as aggregated ones and the results were also consistent with those reported in Table 2.

4.4 Summary of Results

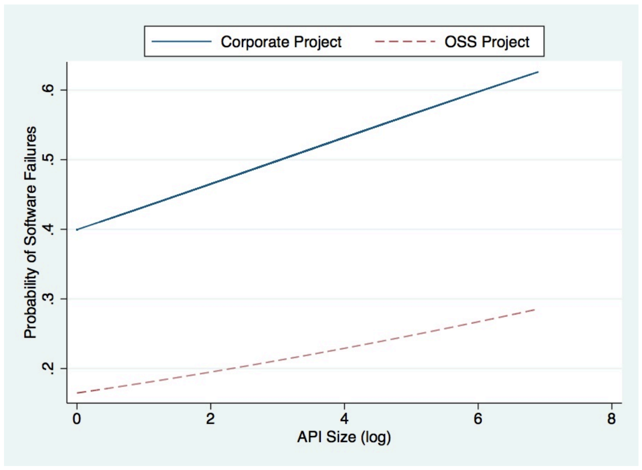
The relevance of the results reported in Table 2 is two-fold. First, the results show that API complexity is an important factor that impacts software failures. Second, the impact of API complexity differs across corporate and open source projects.

Figure 2 depicts these two distinct and valuable results. In this figure, we observe the estimated probability of failure (Y axis) associated with corporate and open source projects as the API complexity increases (X axis). There key issues to highlight. First, corporate projects have a higher initial probability of failure. As discussed in the previous section, the binary variable Corporate Projects captured that effect and the results of the regression indicated that corporate projects were 4.4 times more likely to have failures associated with their source code files than open source projects. Figure 2 also shows the difference in the slopes of the estimated probability curve consistent with the statistically significant effect of the interaction term *API Size X Corporate Project* in model III of Table 2. Corporate projects have a stepper slope associated with them suggesting that the impact of API complexity is more severe in those settings than in the open source projects.

5. DISCUSSION

In this paper, we examined the role of API complexity on software failures across corporate and open source projects. Our first research question focused on the investigation of the impact of API complexity on the failure proneness of source code files. Our results showed that higher levels of API complexity of source code files (measured as sum of the size of the file's exported APIs) are associated with increases in the "buggyness" of files or likelihood of source code files being associated with a defect. We studied the relationship between API complexity and failure proneness in open source and proprietary systems (RQ2). Our analyses suggested that corporate projects tend to have a higher initial probability of failure and are more susceptible to increases in API complexity. More specifically, the impact of API complexity in failure proneness is more severe in corporate projects than in the open source projects.

Figure 2: Estimated Probability of Failure for Corporate and OSS Projects



Our work has several important contributions to the software engineering literature. First, we extended our knowledge about the role of API complexity by examining Bandi and colleagues [2] measures in the context of software failures. Our results are complementary to those of Bandi et al since their results suggested that higher API complexity was positively associated with maintenance time. Second, our analyses combined multiple factors that impact failure proneness. In particular, we extended traditional empirical analyses which focused on churn and product-related metrics with complexity and relational (in the form of syntactic and logical dependencies) characteristics of the APIs. Third and in relation to the debate regarding modularity in open source and proprietary systems (e.g. [27, 33]), our results would suggest that, in fact, open source system are more modular. If we consider modularity in its original conceptual form [32], our results indicating that open source projects are impacted less by API complexity could be interpreted as evidence that complexity tends to be allocated mostly where the functionality is located rather than in the API. Certainly, this is an important area that future research should examine. Finally, we replicated our results across two systems from two different companies as well as nine different open source systems strengthening the external validity of our results.

The remainder of this section discusses the limitations of our work as well as the implications of our results.

5.1 Limitations

Our work has limitations worth noticing. First, the measures proposed by Bandi and colleagues [2] do not consider data elements as APIs. As discussed in the methodology section, we explored two alternative methods of extending the measure and the results were similar. However, we think that redefining the API size measure requires further evaluation. Unfortunately, we did not have access to a third system that also made heavy use of global variables as APIs. Third, recent research has examined the impact of organizational factors on failure proneness such as organizational structure [30] and different types of work dependencies [10]. Again, we did not have access to such data for both systems and therefore cannot use these aspects while conducting our analysis. Finally and as discussed in section 4.2, there was a significant difference in size (in LOCs) between corporate and open source systems. Although our analyses controlled for a wide range of confounding effects, it is possible that the differences in the impact of API complexity are not as significant if we were able to considered larger open source projects. Certainly, it is an important question for future research.

5.2 API Complexity and Software Quality

Software complexity has been an important research topic for several decades [3] and the increasing pervasiveness of software systems suggests that the topic will remain relevant in the future. This line of work has traditionally focused on assessing the complexity of a particular unit of software such as functions, modules or components (e.g. [17]) and examining its impact in the context of software maintenance activities and software quality (e.g [3]). Although an extensive literature has developed over the years, researchers have argued that empirical examinations of the relationship between complexity and quality have produced disappointing results [17]. Our study, on the other hand, examined a dimension of complexity – API complexity - that has been relatively neglected. We are addressing an important gap in the literature because APIs are a central element in modular systems. The modularity literature (e.g. [36]) argues that APIs are the link between modules or components that allows for separation of concerns and development activities, therefore facilitating the coordination among developers. Such argument rests on the assumption that complexity is embedded in the software entity (e.g. module or component) and not in the APIs themselves. Our work and other recent research (e.g. [2, 12, 35]) suggest a departure from that line of work where complexity is in fact embedded in the APIs themselves. Consequently, the separation of technical and work responsibilities is not as simple and pristine as suggested by the modular systems theoretical perspective. Then, complex APIs have the potential to become barriers to effectively decoupling technical responsibilities and work responsibilities and, as consequence, hinder coordination among development teams. The work presented in this paper represents a first step towards characterizing the complexity of APIs and empirically assessing its impact on software quality. The following paragraphs discuss several future research directions we consider will further our understanding of the relationship between API complexity and software quality.

5.2.1 The Nature of API Complexity

The complexity associated with an API relates to multiple dimensions. In this paper and building on prior work, we considered complexity as a function of the number and type of parameters of a particular API. However, there are additional dimensions that future research should explore. First, the set of pre- and post-invocation assumptions made by the API designers and implementers represent also important source of complexity and, consequently, a potential factor leading to failures [31, 34]. For instance, a developer of a module's API has some system configurations for which he/she had developed and validated the particular piece of software. Documentation and communication of these configurations in such a way that it is able to capture all the relevant details is a challenge, leading to a variety of defects when those APIs are utilized. Past research in static analysis [1] has proposed approaches to address problems related with misalignment of assumptions between the API provider and user. However, such research focuses on basic issues such as verification of lock/unlock policies and conformance of particular code structures in device drives. Nambiar and Cataldo [31] presented a set of cases that related to a higher order of misalignment of assumptions that are associated with the information content exchanged through the APIs as well as the environmental context in which the API is expected to operate. Another, but related, dimension of complexity is related to the sequence of invocation of different APIs and the negative consequences that incorrect or unanticipated sequences can have on software quality [25].

Another area that deserves further examination is the implications of particular usage patterns of parameter types for API complexity. For instance, a function or method may take a single integer parameter. In the context of Bandi et al's measures, such API has an operational argument complexity of 1 (the size of the parameter type as defined by [2]). However, the function or method could use such parameter as a mechanism for controlling a complex sequence of statements (e.g. a mask). Arguably, this example implies more complexity in the use of the API because it requires the function or method user to understand a lot more about the API and potentially about the implementation and the implications of particular choices of bits in the mask. This is similar to what Kiczales has termed as “open implementation”. Then, future research should explore ways to capture such differences in complexity into new metrics.

5.2.2 Coordination of Geographically Distributed Development Work

de Souza and Redmiles [12] found that dependencies associated with APIs relating two or more development teams might not be easily identified by those developers. Lack of awareness or disregard for particular dependencies can have important consequences on the quality of a software system. Past research has shown that unsatisfied coordination needs can result in coordination breakdowns, particularly in a geographically distributed settings [9]. These coordination breakdowns tend to materialize as higher number of defects [10]. Our results about the detrimental impact of API complexity on software quality suggest an additional possible source of dependency among the users and the “supplier’ of the APIs that might be difficult to identify and manage. Then, we could consider the complexity of APIs as a factor to help the design of distributed teams that would be better equipped to handle the dependencies imposed by the technical properties of the system. For instance, teams dealing with more complex APIs could be located closer (physically or in terms of time zone difference) than those teams dealing with simpler APIs. In other words, future research should explore the use of metrics that characterize the nature of APIs (e.g. complexity) as an additional mechanism in the identification of what constitutes relevant work dependencies among development teams and organize these teams accordingly. It is important to highlight that recent work has shown that logical dependencies among software modules are a major driver of work dependencies among developers [9]. Our work complements such research by also highlighting the role of the complexity associated with those logical dependencies.

6. CONCLUSION

In this paper, we examine the relative impact of API complexity on the failure proneness of source code files. We performed our analysis using data from two large-scale systems from two distinct software companies and nine different open-source projects. We used Bandi’s et al. API complexity metrics and compared our results with several other predictors of bug failures. Our analyses showed that increases in the complexity of APIs are associated with increases the failure proneness of source code files, i.e., files containing more complex APIs are more likely to have bugs. In addition, the impact of API complexity in failure proneness is more severe in corporate projects than in the open source projects.

7. REFERENCES

[1] T. Ball and S. Rajamani, "Automatically validating temporal safety properties of interfaces," *Model Checking Software,* pp. 102-122, 2001.

[2] R. Bandi, *et al.*, "Predicting maintenance performance using object-oriented design complexity metrics," *Ieee Transactions on Software Engineering,* pp. 77-87, 2003.

[3] R. D. Banker, *et al.*, "Software development practices, software complexity, and software maintenance performance: A field study," *Management Science,* vol. 44, pp. 433-450, 1998.

[4] V. Basili and B. T. Perricone, "Software errors and complexity: an empirical investigation," in *Software engineering metrics I: measures and validations*: McGraw-Hill, Inc., 1993, pp. 168-183.

[5] L. Bass, *et al.*, *Software architecture in practice*, 2nd ed. Boston, MA: Addison-Wesley, 2003.

[6] J. Bloch, "How to design a good API and why it matters," 2006, pp. 506-507.

[7] W. F. Boh, *et al.*, "Learning from Experience in Software Development: A Multilevel Analysis," *Management Science,* vol. 53, pp. 1315-1331, 2007.

[8] L. C. Briand, *et al.*, "Exploring the relationship between design measures and software quality in object-oriented systems," *J. Syst. Softw.,* vol. 51, pp. 245-273, 2000.

[9] M. Cataldo, *et al.*, "Socio-technical congruence: a framework for assessing the impact of technical and work dependencies on software development productivity," presented at the Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement, Kaiserslautern, Germany, 2008.

[10] M. Cataldo, *et al.*, "Software Dependencies, Work Dependencies, and Their Impact on Failures," *IEEE Trans. Softw. Eng.,* vol. 35, pp. 864-878, 2009.

[11] K. Crowston, *et al.*, "Defining open source software project success," 2003, p. 327ñ340.

[12] C. R. B. de Souza and D. F. Redmiles, "On The Roles of APIs in the Coordination of Collaborative Software Development," *Computer Supported Cooperative Work (CSCW),* vol. 18, pp. 445-475, 2009.

[13] J. des Rivieres, "Eclipse APIs: Lines in the sand," *EclipseCon Retrieved March,* vol. 18, p. 2004, 2004.

[14] J. des RiviËres, "How to use the Eclipse API," *Retrieved March,* vol. 9, p. 2004, 2001.

[15] M. Eaddy*, et al.*, "Do Crosscutting Concerns Cause Defects?," *IEEE Trans. Softw. Eng.,* vol. 34, pp. 497-515, 2008.

[16] B. Ellis*, et al.*, "The factory pattern in api design: A usability evaluation," 2007, pp. 302-312.

[17] N. E. Fenton and N. Ohlsson, "Quantitative analysis of faults and failures in a complex software system," *Software Engineering, IEEE Transactions on,* vol. 26, pp. 797-814, 2002.

[18] D. German, "The GNOME project: a case study of open source, global software development," *Software Process Improvement and Practice,* vol. 8, pp. 201-215, 2003.

[19] T. L. Graves*, et al.*, "Predicting fault incidence using software change history," *Software Engineering, IEEE Transactions on,* vol. 26, pp. 653-661, 2000.

[20] R. E. Grinter*, et al.*, "The geography of coordination: dealing with distance in R\&amp;D work," presented at the Proceedings of the international ACM SIGGROUP conference on Supporting group work, Phoenix, Arizona, United States, 1999.

[21] T. Gyimothy*, et al.*, "Empirical validation of object-oriented metrics on open source software for fault prediction," *Software Engineering, IEEE Transactions on,* vol. 31, pp. 897-910, 2005.

[22] M. H. Halstead, *Elements of software science*: Elsevier New York, 1977.

[23] A. E. Hassan and R. C. Holt, "C-REX: an evolutionary code extractor for C," *Submitted for Publication,* 2004.

[24] M. Henning, "API design matters," *Communications of the ACM,* vol. 52, p. 46, 2009.

[25] D. Kawrykow and M. P. Robillard, "Detecting inefficient API usage," 2009, pp. 183-186.

[26] M. H. Kutner*, et al.*, *Applied linear statistical models* vol. 1396: McGraw-Hill Irwin Boston, 2005.

[27] A. MacCormack*, et al.*, "Exploring the Structure of Complex Software Designs: An Empirical Study of Open Source and Proprietary Code," *Management Science,* vol. 52, pp. 1015-1030, 2006.

[28] T. McCabe, "A software complexity measure," *IEEE Trans. Software Engineering,* vol. 2, pp. 308-320, 1976.

[29] N. Nagappan and T. Ball, "Using Software Dependencies and Churn Metrics to Predict Field Failures: An Empirical Case Study," presented at the Proceedings of the First International Symposium on Empirical Software Engineering and Measurement, 2007.

[30] N. Nagappan*, et al.*, "The influence of organizational structure on software quality: an empirical case study," presented at the Proceedings of the 30th international conference on Software engineering, Leipzig, Germany, 2008.

[31] S. Nambiar and M. Cataldo, "Coordination Requirements and Software Architectures: Understanding the Critical Sources of Technical Dependencies," presented at the 2nd International Workshop on Socio-Technical Congruence, Vancouver, Canada, 2009.

[32] D. L. Parnas, "On the criteria to be used in decomposing systems into modules," *Commun. ACM,* vol. 15, pp. 1053-1058, 1972.

[33] J. W. Paulson*, et al.*, "An empirical study of open-source and closed-source software products," *Ieee Transactions on Software Engineering,* pp. 246-256, 2004.

[34] D. E. Perry and W. M. Evangelist, "An Empirical Study of Software API Faults," presented at the International Symposium on New Directions in Computing, Trondheim, Norway, 1985.

[35] C. R. B. d. Souza*, et al.*, "How a good software practice thwarts collaboration: the multiple roles of APIs in software development," presented at the Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering, Newport Beach, CA, USA, 2004.

[36] K. J. Sullivan*, et al.*, "The structure and value of modularity in software design," presented at the Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering, Vienna, Austria, 2001.

[37] R. N. Taylor*, et al.*, "Software Architecture: Foundations, Theory, and Practice," 2009.

[38] P. Wagstrom*, et al.*, "Communication, Team Perforance and the Individual: Bridging Technical Dependencies," presented at the Academy of Management Annual Meeting, Montreal, Canada, 2010.

[39] D. J. Watts, *Small worlds : the dynamics of networks between order and randomness*. Princeton, N.J.: Princeton University Press, 1999.

[40] T. Zimmermann and N. Nagappan, "Predicting defects with program dependencies," presented at the Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement, 2009.