An Analysis of the Effect of Code Ownership on Software Quality across Windows, Eclipse, and Firefox

Christian Bird, Nachiappan Nagappan, Brendan Murphy, Harald Gall, and Premkumar Devanbu

Abstract—Ownership is a key aspect of large-scale software development. We examine the relationship between different ownership measures and software faults/failures in three large software projects drawn from different process domains: Windows Vista, the Eclipse Java IDE, and the Firefox Web Browser. We find that in all cases, measures of ownership such as the number of low-expertise developers, and the proportion of ownership for the top owner have a relationship with both pre-release faults and post-release failures. However, we find that the strength of the effects is related to the development process used. Vista shows the strongest relationship with ownership level, followed by Eclipse, and then Firefox, suggesting that the more that a project uses an open source style process, the more that team sizes rather than ownership levels affect to failures. We also find reasons that low-expertise developers make changes to components and show that the removal of low-expertise contributions dramatically decreases the performance of contribution based defect prediction. Finally we provide recommendations for source code change policies and utilization of resources such as code inspections based on our results.

Index Terms—Empirical Software Engineering, Ownership, Expertise, Defects

1 Introduction

Many recent studies [1]–[4] have shown that human factors play a significant role in the quality of software components. *Ownership* is a general term used to describe whether one person has responsibility for a software component, or if there is no one clearly responsible developer. There is much folklore relating to the effect of ownership on quality. For instance, Raymond claims that in Open Source Software (OSS), despite rather disorganized and dispersed ownership, high quality software is still produced [5]. Within Microsoft, we have found that when more people work on a binary, it has more failures [2], [6]. However, to our knowledge, the effect of ownership has not been studied in depth in commercial and OSS contexts. We suspect that when there is no clear point of contact and the contributions to a software component are spread across many developers, there is an increased chance of communication breakdowns, misaligned goals, inconsistent interfaces and semantics, all leading to lower quality.

Interestingly, unlike aspects of software which are known to be related to defects such as churn, dependency complexity, or size, ownership is something that can be deliberately changed by modifying processes and policies. Thus, the answer to the question: *"How much does ownership affect quality?"* is important as it is *actionable*. Managers and team leads can make better decisions about how to govern a project by knowing the answer. If ownership has a big effect, then policies to enforce strong code ownership can be put into place; managers can also watch out for code which is contributed by developers who have inadequate relevant prior experience. If ownership has little effect, then the normal bottlenecks associated with having one person in charge of each component can be removed, and available talent reassigned at will.

However, our experience is that different projects use different processes, and the effects of ownership may well be related to the development style of a particular project. Specifically, we have observed that many industrial projects encourage high levels of code ownership while open source projects tend to be somewhat more open. In this paper, we examine ownership and software quality in projects with varying software processes and different policies regarding ownership. We make the following contributions in this paper:

1) We present an in depth quantitative study of the effect of multiple measures of ownership on pre-release and post-release defects for multiple large software projects.
2) We deliberately examine projects with different software processes and policies in an effort to see how they are related to the effects of ownership.
3) We identify reasons that components have many low-expertise developers contributing to them.
4) We propose recommendations for dealing with the effects of low ownership in contexts where ownership is strongly related to software quality.

2 Theory & Related Work

A number of prior studies have examined the effect of developer contribution behavior on software quality.

The closest work to ours that we are aware of is that of Weyuker *et al.* [7], that examines the effect of including team size in prediction models. They use a count of the developers that worked on each component, but do not examine the proportion of work, which we account for. They found a negligible increase in failure prediction accuracy when adding team size to their models.

Similarly, Meneely and Williams examined the relationship of the number of developers working on parts of the

Linux kernel with security vulnerabilities [8]. They found that when more than nine developers contribute to a source file, it is sixteen times more likely to include a security vulnerability.

Compared to the current state of the art in code ownership we observe that to the best of our knowledge there has been little or no empirical work on assessing the impact of owners on code quality. New methods like Extreme Programming (XP) [9] profess collective code ownership but there has been no empirical evidence or backing of this data on reasonably mature/complex or large systems. Our study is the first to empirically quantify the effect code owners (and low-expertise contributors) have on the overall code quality.

Domain, application, and even component-specific knowledge are important aids for helping developers to maintin high quality software. Boh *et al.* found that project specific expertise has a much larger impact on the time required to perform development tasks than high levels of diverse experience in unrelated projects [10]. In a qualitative study of 17 commercial software projects, Curtis *et al.* [11] found that "the thin spread of application domain knowledge" was one of the top three salient problems. They also found that one common trait among engineers categorized as "exceptional" was that they had deep domain knowledge, and understood how the system design would generate the system behavior customers expected, even under exceptional circumstances. Such knowledge is not easily obtained. One systems engineer explained, "Someone had to spend a hundred million to put that knowledge in my head. It didn't come free."

The question naturally arises, how can we determine who has such domain knowledge? Fortunately, there is a wealth of literature that uses the prior development activity on a component as a proxy for expertise and knowledge with respect to the component. As examples Expertise Browser from Mockus *et al.* [12] and Expertise Recommender from McDonald and Ackerman [13] both use measures of the amount of work that a developer has performed on a software component to recommend component experts. Fritz *et al.* found that the ability of a developer to answer questions about a piece of code in a system was strongly determined by whether the developer had authored some of the code, and how much time was spent authoring it [14]. Mockus and Weiss found that changes made by developers that were more experienced with a piece of code were less likely to induce failures [15]. In a study of offshoring and succession in software development [16], Mockus evaluated a number of succession measures with the goal of being able to automatically identify mentors for developers working on a per-component basis. A succession measure based on ownership was able to accurately pinpoint the most likely method and was used in a large scale study evaluating the factors affecting productivity in project succession and offshoring.

Research in other domains, such as manufacturing, has found that when a worker performs a task repeatedly, the labor requirements to complete subsequent work in the same task decreases and the quality increases [17]. Software development differs from these domains in that workers do not perform the exact same task repeatedly. Rather, software development represents a form of constant problem solving in which tasks are rarely exactly the same, but may be similar. Nonetheless, developers gain project and component specific knowledge as they repeatedly perform tasks on the same systems [18]. Banker *et al.* found that increased experience increases a developer's knowledge of the architectural domain of the system [19]. Repeatedly using a particular API, or working on a particular system creates *episodic knowledge*. Robillard indicates that the lack of such knowledge negatively affects the quality of software [20]. Indeed, Basili and Caldiera present an approach for improving quality in software development through learning and experience by establishing "experience factories" [21]. They claim that by reusing knowledge, products, and experience, companies can maintain high quality levels because developers do not need to constantly acquire new knowledge and expertise as they work on different projects. Drawing on these ideas, we develop ownership measures which consider the number of times that a developer works on a particular component, with the idea that each exposure is a learning experience and increases the developer's knowledge and abilities.

There is a knowledge-sharing factor at play as well. The set of developers that contribute to a component implicitly form a team that has shared knowledge regarding the semantics and design of the component. Coordination is a known problem in software development [22]. In fact, another of the top three problems identified in Curtis' study [11] was "communication and coordination breakdowns." Working in such a group always creates a need for sharing and integrating knowledge across all members [23]. Cataldo *et al.* showed that communication breakdowns delay tasks [1] If a member of this team devotes little attention to the team and/or the component, they may not acquire the knowledge required to make changes to the component without error. We attempt to operationalize these team members in this paper and examine their effect on quality.

If ownership of a particular component in a system (whether it be a file, class, module, plugin, or subsystem) is a valid proxy for expertise, then what is the effect of having most changes made by those with little expertise? Is it better to have one clear owner of a software component? We operationalize ownership in two key ways here and formally define our measures in section 3. One measure of ownership is how much of the development activity for a component comes from one developer. If one developer makes 80% of the changes to a component, then we say that the component has high ownership. The other way that we measure ownership is by determining how many low-expertise developers are working on a component. If many developers are all making few changes to a component, then there are many non-experts working on the component and we label the component as having low ownership.

We expect that having one clear "owner" for a component will lead to fewer failures and that when many non-experts are making changes, indicating that ownership is spread across many contributors, the component will have more failures.

Based on our observations and prior work [24], the management and social organization in an open source project is more free form than commercial projects, though clearly still existent, and developers are less constrained to work in only one portion of the code base (and also, are not held as responsible for a particular component). Thus, we expect that the strength of the relationship between ownership and software quality may differ based on the development style. We therefore examine ownership in multiple projects using different styles.

We expect that the relationship of ownership with code quality will differ with development process.

3 Terminology and Metrics

We adopt Basili's goal question metric approach [25] to frame our study of ownership. Our goal is to understand the relationship between ownership and software quality. We also hope to gain an understanding of how this relationship varies with the development process in use. Achievement of this goal can lead to more informed development decisions or possibly process policy changes resulting in software with fewer defects.

In order to reach this goal, we ask a number of specific questions:

1) Are higher levels of ownership associated with less defects?
2) Is there a negative effect when a software entity is developed by many people with low ownership?
3) Are these effects related to the development process used?

In order to answer these questions, we propose a number of ownership metrics and use them to evaluate our hypotheses of ownership. We begin by defining some important terms and metrics used throughout the rest of this paper:

- Software Component – This is a unit of development that has some core functionality. Defects can be traced back to a specific component and software changes from developers can also be traced to a component. In Windows Vista, a component is a compiled binary (.exe, .dll, .sys). In Eclipse this is a plugin or top level package. In Firefox, this represents a set of Java or C++ files in the same directory, (as used in prior studies [26]).
- **Contributor** – A contributor to a software component is someone who has made commits/software changes to the component.
- **Proportion of Ownership** – The proportion of ownership (or simply ownership) of a contributor for a particular component is the ratio of number of commits that the contributor has made relative to the total number of commits for that component. Thus, if Cindy has made 20 commits to `org.eclipse.jdt.ui` and there are a total of 100 commits to `org.eclipse.jdt.ui` then Cindy has an ownership of 20%.
- **Minor Contributor** – A developer who has made changes to a component, but whose ownership is below 5% is considered a minor contributor to that component. This threshold was chosen based on examination of distributions of ownership[1]. We refer to a commit from a minor contributor as a minor contribution.
- **Major Contributor** – A developer who has made changes to a component and whose ownership is at or above 5% is a major contributor to the component and a commit from such a developer is a major contribution.

With these terms defined, we now introduce our metrics.

- Minor – number of minor contributors
- Major – number of major contributors
- Total – total number of contributors
- Ownership – proportion of ownership for the contributor with the highest proportion of ownership

Figure 1 shows the proportion of commits for each of the developers that contributed to `abocomp.dll` in Windows Vista, in decreasing order. This library had a total of 918 commits made during the development cycle. The top contributing engineer made 379 commits, roughly 41%. Five engineers made at least 5% of the commits (at least 46 commits). Twelve engineers made less than 5% of the commits (less than 46 commits). Finally, there were a total of seventeen engineers that made commits to the binary. Thus, our metrics for `abocomp.dll` are:

Metric	Value
Minor	12
Major	5
Total	17
Ownership	0.41

4 Hypotheses

We begin with the observation that a developer with lower expertise is more likely to introduce bugs into the code. A developer who has made a small proportion of the commits to a binary likely has less expertise and is more likely to make an error. We expect that as the number of developers working on a component increases, the component may become "fragmented" and the difficulty of vetting and coordinating all these minor contributions becomes an obstacle to good quality. Thus if Minor is high, quality suffers.

Hypothesis 1 - Software components with many minor contributors will have more failures than a software components that have fewer.

1. A sensitivity analysis with threshold values ranging from 2% to 10% yielded similar results.

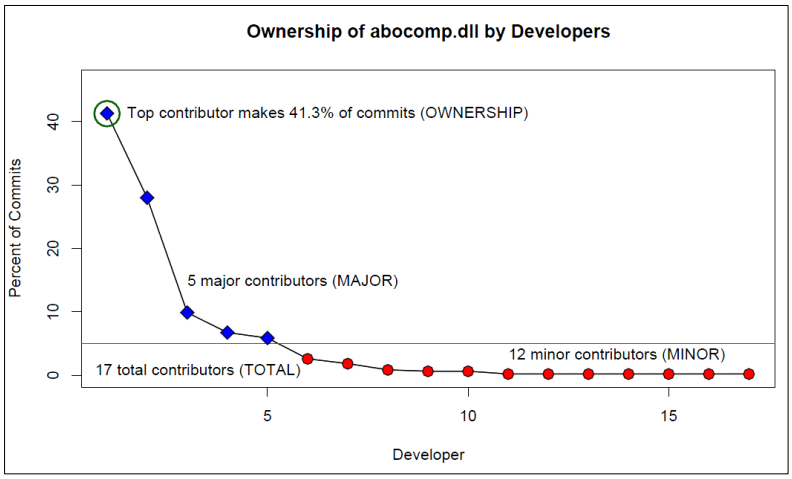


Fig. 1: Graph of the proportion of commits to `abocamp.dll` by developers during the Vista development cycle, showing the four measures of ownership used in this paper.

We also look at the proportion of ownership for the highest contributing developer for each component (OWNERSHIP). If OWNERSHIP is high, that indicates that there is one developer who "owns" the component and has a high level of expertise. This person can also act as a single point of contact for others who need to use the component, need changes to it, or just have questions about it. We theorize that when such a person exists, the software quality is higher resulting in fewer failures.

Hypothesis 2 - Software components with a high level of ownership will have fewer failures than components with lower top ownership levels.

We also expect that the relationship between our measurements of ownership will vary with the development process used. Corporations like Microsoft, where development is more controlled, require developers to work in specific portions of a large project: this type of work assignment is key to the governance of large teams, while maintaining accountability and responsibility. In most OSS projects, contributors have more freedom to choose their work assignments and cannot so easily be directed to work on a specific portion of the project. While Raymond suggests that OSS has a bazaar-like organization [5], we have observed that many of the large OSS projects are in fact organized (though still not as rigidly as in a corporate setting) and that different participants tend to gravitate towards specific parts of the system [24]. Thus, although the OSS projects in our study do not have clear ownership policies and guidelines as commonly found in industry, there *are* gatekeepers for certain parts of the system that make decisions and vet changes. It is certainly true that many open source developers are in fact paid to work on particular projects by their employers; however, rarely does a single employer maintain full control of the project, and, in fact, a single project may have contributors that are sponsored by many employers. ECLIPSE represents a bit of an anomaly in that most contributors are employed by one entity, IBM. We study the "core" ECLIPSE development environment and do not include third-party plugins.

Because there is little externally imposed organization, or explicit code ownership policies, with OSS projects, developers are less constrained to work in only one portion of the code base; also, they are not held as responsible for a particular component. Therefore, in the OSS setting, we expect that the strength of the relationship between ownership and software quality will be lower than in corporate settings.

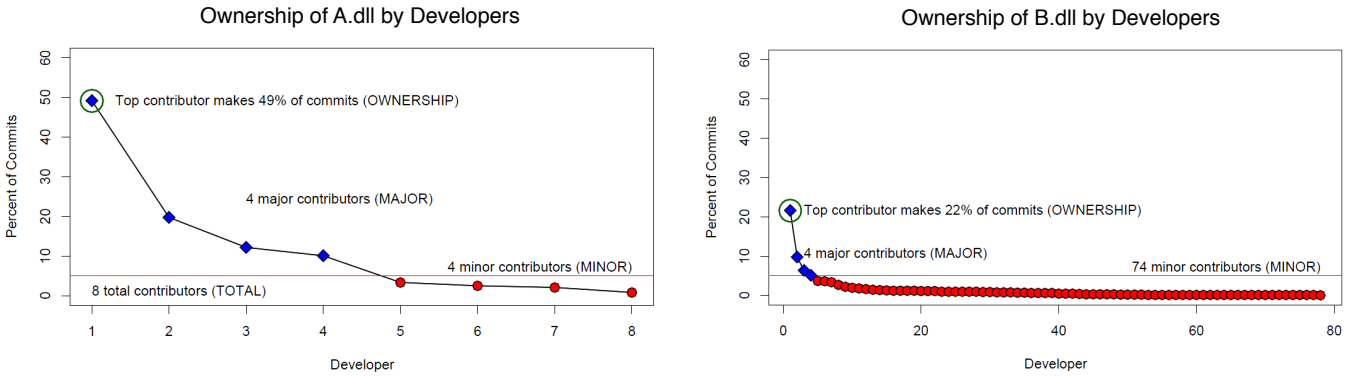
Hypothesis 3 - The relationship between ownership and number of software failures will be lower in software projects that use a more open source style development process.

We take no position on the merits of any development style; we simply theorize that the effect of ownership on software quality changes as a result of the development process.

5 Data Collection and Analysis

This data presents an opportunity to investigate hypotheses regarding code ownership. In addition, there are also large and successful Open Source Software (OSS) projects that make their data readily available for study. Gathering data from multiple projects that differ in development style enables us to examine, qualitatively, whether the relationship between ownership and software quality is a general phenomenon, or if it is process dependent. In this study, we examine Windows Vista, the ECLIPSE Java IDE, and the FIREFOX Web Browser.

Windows Vista is controlled and developed completely by Microsoft, who have processes and policies that favor strong code ownership. Vista was developed by 2,000+ software developers and is composed of thousands of individual executable files (`.exe`), shared libraries (`.dll`), and drivers



(a) A.dll (b) B.dll

Fig. 2: Ownership graphs for two binaries in Windows Vista

Fig. 3: Spectrum of development processes from commercial to open source.

(`.sys`), which we collectively refer to as *binaries*. We track the development history from the release of Windows Server 2003 to the release of Windows Vista and include pre-release defects as well as post-release failures in the first six months after release as software quality indicators.

The ECLIPSE *Java IDE* is a project developed by IBM. Although it is technically open source, many of the commits to it come from IBM employees and IBM appears to govern much of the project. It thus represents a "hybrid" project that is largely controlled by one entity, but which espouses open source principles, making the source code available under an open source license (the ECLIPSE Public License). ECLIPSE conducts work in the open, accepting contributions and in some cases, guidance, from the larger community. Although external contributions are common, discussions with core members indicate that these are usually small patches that fix issues rather than complete additional features, and often require manual editing by ECLIPSE developers prior to being accepted into the code base [27]. We track development activity as well as pre- and post-release bugs for six major releases: 2.0, 2.1, 3.0, 3.1, 3.2, and 3.3.

The *Firefox web browser* is more representative of the open source ecosystem, though clearly there is large variance in the types of OSS projects and we do not claim that there is *one* style of OSS development. Contributions to FIREFOX come from a myriad of sources and no single commercial entity completely controls or owns the development process[2]. We track development activity as well as pre and post-release bugs for two major releases of Firefox: Release 1.5 (November 2005) and 2.0 (October 2006). During this time, over 500 paid and unpaid people contributed to the browser.

These three projects form a continuum in the process domain (Figure 3). Vista is closed source, has rigorous ownership policies and is wholly controlled by a single company. Eclipse is more open, accepts contributions from external entities, and follows a more open source development methodology, but is still mostly controlled by IBM. Firefox is a more "traditional" open source project with repository access open to anyone who demonstrates competent skills and a desire to contribute to the project.

We require several types of data. The most important data are the commit histories and software failures. Software repositories record the contents of every change made to a piece of software, along with the change author, the time of change, and an associated log message that may be indicative of the type of change (e.g. introducing a feature, or fixing a bug). For Windows Vista, we collected the number of changes made by each developer to each source file and used a mapping of source files to binaries in order to determine the number of changes made by each developer to each binary. For ECLIPSE and FIREFOX, we mined the changes from the publicly available CVS repositories. For ECLIPSE, we use *plugins* as the level of granularity for software components (specifically, JAR files within plugins, as some plugins are composed of multiple JAR files) because the quality of the defect data is not as reliable at the source file level due to the low number of defects that most source code files have. In addition the majority of bug fixes are tied to multiple source files, but the same plugin. The number of plugin JAR files per release ranged from 90 in 2.0 to 250 in 3.3. We used only the plugins that are

2. Although Firefox is still contributed to by people from many companies, Mozilla Corp now maintains more control. This was not as true for the historical releases that we examined.

part the official ECLIPSE project and exist in the ECLIPSE CVS repository. For Firefox, we examined components by directory structure as used in prior work [26] for similar reasons. A manual inspection indicated that subsystems within firefox are reflected in the directory structure.

We used the commit logs and mapping of source files to software components to categorize contributors into minor and major contributors on a software component basis and calculate the ownership level for each component.

We also gathered both pre-release and post-release software failures for all three projects. For Vista, we gathered the failures recorded prior to release and in the first six months after release. Because of the information contained in the failures, we can automatically trace them back to the binaries that caused them. For ECLIPSE, we mined the bug database and used automated techniques to link closed bugs in the database to the commits that fixed them in the software repository [28], [29]. Each bug in the database includes the date when it was opened and also the version of Eclipse that the bug occurred in. We use this information along with the release dates of each version of Eclipse to categorize bugs into pre-release and post-release. Of necessity, this only includes bugs that were a) actually fixed and b) manually attributed in the log message so that an automated process could identify it. We argue that the most important bugs are those that are fixed and thus those that matter the most are being recorded. In addition, Eclipse has a rigorous policy of manually attributing bug IDs in their log messages. For Firefox, the pre and post-release data for the 1.5 and 2.0 releases was manually obtained by an open source engineer who was contracted for the purpose of this research study to obtain the best possible data. We therefore have high confidence in the failure information. This developer also annotated each bug with whether it was a pre-release or post-release failure. For all three projects, we only count failures that the development team deemed important enough to fix.

Finally, we gathered source code metrics including various size, complexity, and churn metrics. For Vista, we gather this information from the source code repositories and the build process. For Eclipse and Firefox, we gathered churn from the repository history and size and complexity by using UNDERSTAND[3], a tool from SciTools which calculates a number of metrics on C, C++, and Java code.

5.1 Analysis Techniques

We use a number of methods to examine the relationship between ownership and software quality.

We began with a correlation analysis of both pre and post-release failures with each of the ownership metrics as well as a number of other metrics such as test coverage, complexity, size, dependencies, and churn. The results indicated that pre and post-release defects had strong relationships with MINOR, TOTAL, and OWNERSHIP. In fact, *in Windows Vista,* MINOR *had a higher correlation with both pre (0.86) and post-release defects (0.70) than any other metric that Microsoft collect!*

However, we also observed some relationship between code attributes and ownership metrics. For example, Figure 2 shows data for two binaries within Vista with vastly different ownership profiles. Unsurprisingly, the binary depicted in Figure 2-b (`B.dll`) has more failures than the binary in Figure 2-a (`A.dll`), eight times as many pre-release failures and twice as many post-release failures. However, `B.dll` is also a larger binary and experienced far more churn during the Vista development cycle. Thus it is not clear whether the increase in failures is attributable to more minor contributors or other measures such as size, complexity, and churn, which are known to be related to defects [30], [31] and are likely related to the number of minor contributors. Prior research has shown that when characteristics such as size are not considered, they may affect the validity of other software metrics [32].

To overcome this problem, we used multiple linear regression, which is primarily used in two different ways. First, it can be used to make predictions about an outcome based on prior data (for instance predicting how many failures a software component may have based on characteristics of the components). Second, it enables us to examine the effect of one or more variables when controlling for other variables. We use it for the latter in an effort to examine the relationship of our ownership measures *when controlling for source code characteristics such as size, complexity, and churn.* The linear regression model for failures indicates which variables have an effect on failures, how large the effect is, in what direction (i.e. if failures go up when a metric goes up or when it goes down), and how much of the variance in the number of failures is explained by the metrics. We compare the amount of variance in failures explained by a model that includes the ownership metrics to a model that does not include them. There are many measures of churn, complexity, and size. However, to avoid multi-collinearity, we include only one of each measure in the model; We choose the measure which results in the best base model. This gives an indication of how much ownership actually affects software failures. We examined the improvement in amount of variance in failures explained by the metrics (commonly referred to as the adjusted R^2) and examine improved goodness of fit using F-tests to determine if the addition of an ownership metric improves the model by a statistically significant degree [33].

Linear regression models can be reliably interpreted if certain assumptions hold. Two key assumptions are that the residuals are normally distributed, and not correlated with any of the independent variables. In our analysis, we found that the distribution of failures was almost always heavily right skewed. When we transformed the dependent variable to be the `log` of the number of failures, the skew diminished, and the residuals reasonably fit the normality assumption. This data transformation was applied to all dependent variables except for post-release failures in Vista, where linear regression assumptions were met by the raw data.

3. http://www.scitools.com/products/understand

Windows Vista

Model	Pre-Release Failures	Post-Release Failures
Base (code metrics)	26%	29%
Base + MINOR	46%*(+20%)	41%*(+12%)
Base + TOTAL	40%*(+14%)	35%*(+6%)
Base + MINOR + OWNERSHIP	50%*(+4%)	44%*(+3%)

TABLE 1: Variance in failures for the base model which includes standard metrics of complexity, size, and churn, as well as the models with MINOR and OWNERSHIP added. An asterisk* denotes that a model showed statistically significant improvement when the additional variable was added.

Eclipse 3.0

Model	Pre-Release Failures	Post-Release Failures
Base (code metrics)	35%	40%
Base + MINOR	44%*(+9%)	48%*(+8%)
Base + TOTAL	46%*(+11%)	49%*(+9%)
Base + MINOR + OWNERSHIP	48%*(+4%)	51%*(+3%)
Base + TOTAL + OWNERSHIP	47% (+1%)	51%*(+2%)

TABLE 3: Variance in failures explained for ECLIPSE regression models.

6 Results

We now present and contrast the results of our analysis of Windows Vista, ECLIPSE and FIREFOX. Tables 1 and 2 illustrate the results of our analysis. We denote with an asterisk*, cases where an goodness-of-fit F-test indicated that the addition of a variable improved the model by a statistically significant degree. The value in parentheses indicates the percent increase in variance explained over the previous model (the model without the additional variable added). For example, in Table 1 the Base+Minor+Ownership model in Vista explains 50% of the variance in pre-release failures which is 4% more than the Base+Minor model which explains 46%.

6.1 Windows Vista

We built four statistical models of failures for Windows Vista (summarized in Table 1). The first model contains only the classical source code metrics: size, complexity, and churn. We refer to this as the base model. This model showed that churn, size, and complexity all have a statistically significant effect on both pre and post-release failures. In addition, these metrics are able to explain 26% of the variance in pre-release failures and 29% of the variance in post-release failures.

In the second model, we added MINOR to the set of predictor variables in the base model. The statistics showed that MINOR is positively related to both pre and post-release failures to a statistically significant degree. The addition of MINOR increased the proportion of variance in pre-release failures to 46% and post-release failures to 41%.

Next, we included TOTAL instead of MINOR in addition to the classic variables. This was performed to see if the total number of developers has a different effect on quality than the number of minor contributors. The gains shown by MINOR were stronger than those shown by TOTAL for both types of failures to a statistically significant degree, indicating that MINOR has a larger effect on failures.

Finally, we added OWNERSHIP, the level of ownership for the top contributor, to the second model, which contained the base code measures and MINOR. OWNERSHIP was found to have a negative relationship with failures to a statistically significant degree. The addition of this variable increased the explanatory power of the model, but not as much as the addition of MINOR. We found this despite the order of adding variables to the model. MINOR still showed more of an effect than OWNERSHIP even when it was added second (not shown). This model explains 50% of variance in pre-release failures and 44% of variance in post-release failures; We also observed a similar marginal increase in variance explained when adding OWNERSHIP to the base model.

The results of our analysis of ownership in Windows Vista can be interpreted as follows:

1) *The number of minor contributors has a strong positive relationship with both pre- and post-release failures even when controlling for metrics such as size, churn, and complexity.*
2) *Higher levels of ownership for the top contributor to a component results in fewer failures when controlling for the same metrics, but the effect is smaller than the number of minor contributors.*
3) *Ownership has a stronger relationship with pre-release failures than post-release failures.*

6.2 Eclipse

The same regression analysis was conducted on pre- and post-release defects across six major releases of ECLIPSE. Table 3 contains our results from only one release of ECLIPSE, chosen because it was most representative of our findings[4] We describe the differences in our ECLIPSE models below.

In ECLIPSE, the addition of MINOR to the model always improved the model's explanatory power by 4-10% with regard to pre-release failures and the increase was always statistically significant. The improvements were between 1 and 8% for post-release failures. However, in many cases, there was no statistically significant difference between adding MINOR to the model and adding TOTAL, and in a few cases, TOTAL was better to a statistically significant degree. Because both MINOR and TOTAL had large effects, but neither was consistently better, we built regression models for both which also incorporated OWNERSHIP (last two models in Table 3). This addition resulted in a marginal

4. The complete findings can be found at http://wwwcsif.cs.ucdavis.edu/~bird/ownership.

Firefox 1.5 & 2.0

Release	Model	Pre-Release Failures	Post-Release Failures
1.5	Base (code metrics)	32%	32%
	Base + MINOR	53%*(+21%)	36%*(+4%)
	Base + TOTAL	60%*(+28%)	37%*(+5%)
	Base + MINOR + OWNERSHIP	58%*(+8%)	37% (+1%)
2.0	Base (code metrics)	35%	23%
	Base + MINOR	47%*(+12%)	26%*(+3%)
	Base + TOTAL	51%*(+16%)	27%*(+4%)
	Base + MINOR + OWNERSHIP	50%*(+3%)	27% (+1%)

TABLE 2: Variance in failures explained for Firefox for the base model which includes standard metrics of complexity, size, and churn, as well as the models with MINOR, TOTAL and OWNERSHIP added.

improvement of 1-3% and in six of the 24 cases (we added OWNERSHIP to the TOTAL and MINOR models for pre- and post-release failures for all six releases) the effect was not statistically significant. In every case where there was a statistically significant effect, higher values for MINOR and TOTAL increased failures and higher levels of OWNERSHIP reduced them.

We summarize our findings:

1) *Both* MINOR *and* TOTAL *have a positive relationship with pre and post-release defects. However, neither is consistently a better indicator, and the effect is weaker than in Vista.*
2) *Higher levels of* OWNERSHIP *sometimes have a positive relationship with pre and post-release quality, but the effect is small when it is statistically significant.*
3) *Ownership measures have a slightly larger effect on pre-release failures than post-release failures.*

6.3 Firefox

We also performed our regression analysis on the 1.5 and 2.0 releases of Firefox yielded results that differ from the other projects. In both cases, we examined both pre- and post-release failures and built a series of models to examine the effect of ownership when controlling for size, churn and complexity. Table 2 show our results for FIREFOX.

As before, the addition of both MINOR and TOTAL to the base model resulted in a strong improvement in the model. However, unlike the other projects, TOTAL had a stronger relationship with defects than MINOR *for all data sets*. The effect was large, from 12% to 20% for pre-release defects and 4% for post-release. Furthermore, the addition of OWNERSHIP to the MINOR models only showed a statistically significant improvement in the pre-release defects, though even this augmented model was never better than the TOTAL model. TOTAL is more related to the number of people working on a component than their level of expertise.

We make the following key observations from the Firefox data:

1) *Team size has a stronger relationship with defects than ownership levels.*
2) *Team size and ownership metrics have a much stronger relationship with pre-release defects than post-release defects.*

6.4 Comparison

We summarize the comparison of results for our analysis of pre-release and post-release defects in Vista, ECLIPSE, and FIREFOX in Table 4. This table characterizes the magnitude from *none* to *strong* based on the magnitude of the effect of each metric relative to the others. For metrics that measure ownership levels (MINOR and OWNERSHIP), there is a clear trend of being stronger in Vista than in ECLIPSE and stronger in ECLIPSE than FIREFOX. In all cases, the effect of MAJOR was weak and often not statistically significant, indicating that the number of higher-expertise contributors has little effect on quality. The total number of contributors has an effect in all projects, but a stronger effect in the more open source style projects that do not have enforced ownership policies. In the context of Vista, where formal ownership policies are in place, the violation or adherence to such policies have an effect on software quality. In the two projects without such policies, we see an effect, but it is clearly not as strong.

7 Effects of Minor Contributors

One of the key findings in our analysis was that the number of minor contributors has a strong relationship with failures in Windows Vista, but a weaker relationship in ECLIPSE and FIREFOX. Since the effect was strongest in Vista, and because Microsoft could make changes to practices based on these findings, we were eager to gain a deeper understanding of this phenomenon in that particular dataset; so we performed two more detailed analyses.

First, we observed that almost all developers were major contributors to some binaries and minor contributors to others. This led us to investigate the obvious question: *Given a particular developer, is there a relationship between a component to which she is a major contributor, and the one to which she is a minor contributor?*

Metric	Windows Vista	Eclipse	Firefox
Code Metrics (Base)	Medium +	Medium to Strong +	Medium +
Minor	Strong +	Medium to Strong +	Medium +
Major	Weak +	Weak +	Weak to None +
Ownership	Medium −	Weak −	None

TABLE 4: Summary of relationship of Ownership metrics with failures. A + indicates that failures increased with higher values of a metric and a − indicates that failures decreased.

Second, we adapted a fault prediction study carried out by Pinzger *et al.* [3] and examined the effect of modifying the study in ways related to ownership.

7.1 Dependency Analysis

The majority of developers that contributed to Vista acted as major contributors to some binaries and minor contributors to others. There are very few Vista developers who are only minor contributors. This fact is an indication of strong code ownership, as it shows that nearly everyone has a main responsibility for at least one binary.

Discussions with engineers at Microsoft indicated that often an engineer who was the owner of one binary would make changes to another binary whose services he or she used, often in the process of addressing reported bugs. In our context this would show up as one engineer who was a major contributor to some binary, A, and a minor contributor to some binary, B, with a dependency relationship between A and B. We call this a *Major-Minor-Dependency* relationship, which is illustrated in Figure 4.

Cataldo *et al.* found that making changes to a depending component without coordinating with the other stakeholders (in our case, the owner) of the component increases the likelihood of faults [1]. We have no record of the communication between developers of Windows Vista. However, the fact that a minor contributor has, by definition, made few if any prior contributions to a component suggests that their participation in the component's implicit team is likely minimal, increasing the risk of a introducing a bug.

But does this actually happen? Is a developer D, working on binary *Foo.exe*, statistically more likely to be a minor contributor to a binary *Bar.dll*, just because *Foo.exe* depends on *Bar.dll*? If so, how many of the minor contributors to components can this phenomenon account for? If the majority of minor contributors are a result of component owners making changes do depending or dependent components to accomplish their own tasks such as resolving failures, then deliberate steps could be taken to avoid this type of risky behavior.

To investigate this further, we first used MaX, a static analysis tool at Microsoft to detect dependency relationships between binaries [34]. These relationships include method calls, read and writes to the registry, IPC, COM calls, and use of types. Using this tool, we construct a dependency graph that includes all of the binaries in Windows Vista.

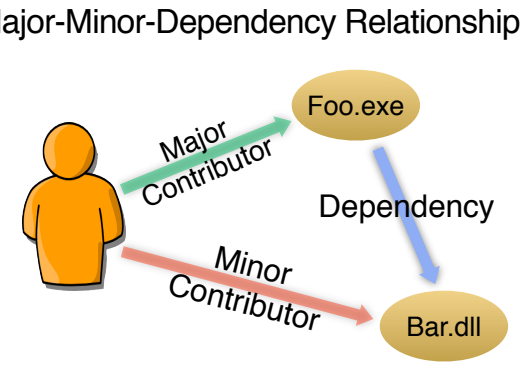
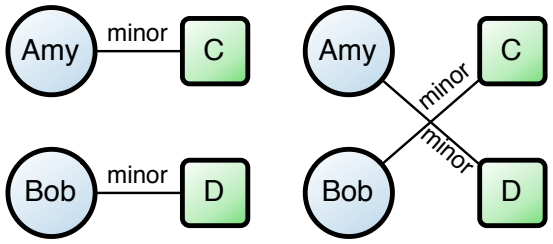


Fig. 4: Illustration of the major-minor-dependency relationship commonly observed in Vista

The next step is to determine whether the major-minor-dependency phenomenon occurs statistically more often than would be expected by chance. But what exactly does "by chance" mean? We model "chance" by generating a large, plausible, random sample of contributions; we can then compare the *observed* frequency of major-minor dependency with the frequency in the *generated* sample. Our plausible random model is that each developer chooses their contributions at random, while preserving their rate of minor and major contributions. In other words, a developer is just as hardworking, but her choice of where to contribute is not influenced by dependencies in the code. Using this model, we generate a large sample of simulated contribution graphs. This gives us a basis for comparison to evaluate observed, real-world contribution behavior is influenced by dependencies between modules.

This "bootstrapping" approach comes from the statistical theory of random graphs [35]–[37]. A phenomenon is judged statistically significant if the actual, observed phenomenon occurs rarely in the generated sample graphs. Following previous techniques [35], [36], we use a *graph-rewiring* method to bootstrap our random ensemble, based on the observed frequency of commits from individuals. In each generated random sample, each developer makes the same number of major and minor contributions as in the observed real sample, but the contributions are chosen at random from the given set of components. We check to see how often a "majorly contributed component" has an actual dependency on a "minorly contributed component"



(a) before (b) after

Fig. 5: An illustration of graph rewiring. Rewiring preserves the number of minor and major edges per developer and per binary, but randomizes the organization of the contribution network.

Release	Actual %	Mean Random %	p-value
2.0	62.0%	35.4%	$\ll .01\%^*$
2.1	23.3%	21.8%	0.333%
3.0	31.3%	29.1%	0.224%
3.1	46.1%	31.3%	$\ll .01\%^*$
3.2	48.4%	41.6%	$0.016\%^*$
3.3	47.8%	36.4%	$0.001\%^*$

TABLE 5: The actual proportion of binaries with major-minor-dependencies as well as the mean of 100,000 random graphs with the same major and minor edge distributions for six releases of Eclipse. The p-value is the likelihood that the actual proportion is no higher than the proportion in the random graphs. * indicates statistically significant results.

in these generated random samples. If the frequency of major-minor-dependency relationships that occur in the ensemble of simulated samples differs significantly from that of the observed real sample, then we can conclude that the phenomenon most likely represents some real, intended behavior, and not simply a chance occurrence.

Graph rewiring is performed as follows in our context. For the sake of convenience we refer to an edge connecting a binary to one of its major contributors as a *major edge* and an edge connecting a binary to one if its minor contributors as a *minor edge*. Two edges that are either both major edges or both minor edges are selected at random and endpoints of both are switched. Thus, after the switch, the number of major and major contributions for each developer node and each component node remains the same.

After performing E^2 rewirings where E is the number of contribution edges in the graph, a sufficiently random graph is obtained. We created 10,000 such random contribution graphs and compared the frequency of major-minor-dependency relationships to the frequency in the observed, actual contribution graph. We found that in the observed Vista contribution graph, 52% of the binaries had minor contributors who were major contributors to other binaries that the original had a dependency with. In contrast, this relationship only existed for an average of 24% of the binaries in the random networks with the same minor and major contribution degree distributions. The maximum value for the normally distributed frequency of this phenomenon out of all 10,000 graphs was 32% of the time, indicating that 52% is definitely a statistically significant difference, and the phenomenon that we are observing does not occur by chance.

In Vista, one common reason that a developer is a minor contributor to a binary is that he/she is a major contributor to a depending binary. This allows for processes to be put into place to recognize and either minimize or aid minor contributions

We also performed this analysis on our Eclipse data. We used the static analysis tool **Understand** from SciTools[5] to identify program dependencies within Eclipse. The dependencies are determined at the class level, and we use a mapping from classes to files and then to plugins to determine plugin dependencies. A class A may depend on a class or interface B if A inherits or implements B, has a field of type B, calls a method in B, imports B, creates an instance of B, or contains a method that either returns or accepts a parameter of type B.

For each release of Eclipse, we compared the number of binaries that had major-minor-dependency relationships in the actual graph with 100,000 randomly generated graphs with the same dependencies and the same major and minor contribution edge distributions.

The difference between the random graphs and the actual software graphs made of real major and minor contribution edges and dependencies is significant in four of the six Eclipse releases. However, even in cases where the difference is statistically significant, the magnitude of the difference is not large; only the 2.0 release shows a difference of more than 15%. This contrasts sharply with our results for Vista. This indicates that although the major-minor-dependency relationship does occur in Eclipse more that would be expected by chance, it is not as strong of a phenomenon here as in Vista and likely does not explain the reason that some plugins have many minor contributors while others do not. This gives further evidence that ownership does not play as strong a role in Eclipse as it does in Vista, where almost half of the minor contributors (violators of strong ownership practices) were explained by dependency relationships.

The major-minor-dependency relationship is statistically significant in Eclipse*, but occurs less than is Vista, indicating that the reasons for minor contributors differ between these projects.*

We were unable to perform this analysis on the Firefox code base. The many different languages in use (e.g. C, C++, javascript, XUL, XML) in this project make identifying dependencies difficult.

7.2 Effects on Network Metrics

In 2007, Pinzger *et al.* reported a method to find fault prone binaries in Windows Vista based on contribution networks [3]. A contribution network is composed of

5. http://www.scitools.com/products/understand

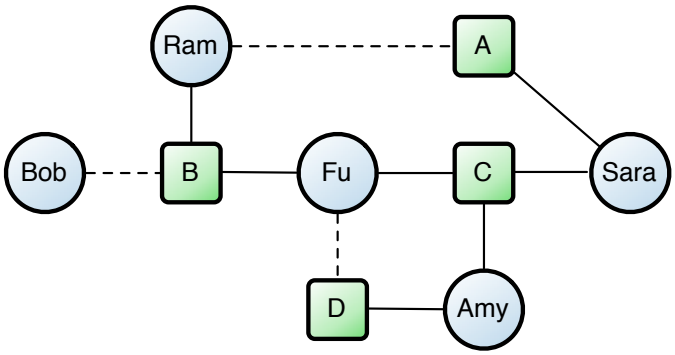


Fig. 6: An example contribution network. Boxes represent binaries and circles represent developers who contributed to them. A dashed line between a binary and developer indicates a minor contributor relationship.

binaries and the developers that contributed to those binaries. Thus, a node representing a developer is connected to all binaries the developer has contributed to and a node representing a binary is connected to all developers that contributed to it. Figure 6 shows an example of a contribution network with boxes representing binaries and circles representing developers. Major contribution edges are solid and minor contribution edges are dashed.

The field of social network analysis has developed a number of metrics that measure various attributes of nodes in a network. For instance, the degree of a node is the number of direct connections that it has and can be indicative of how important the node is within the local network. Other metrics measure how much information can flow through a node, the average distance from a node to all other nodes, how much "power" a node exerts over its neighbors, etc. An in-depth discussion of these metrics can be found in Wasserman and Faust [38]. Pinzger *et al.* found that these measures had a strong relationship with post-release failures in Windows Vista and in a companion study [4] we found that these measures were able to predict failures in Eclipse accurately as well.

Specifically, when they built a predictor for fault prone binaries using this method, it identified 90% of the fault prone binaries in Vista (recall) and 85% of the binaries that it classified as fault prone actually were (precision). This was a dramatic increase over the predictive power of prior methods that used source code metrics.

We adapted that study, and examined the effect of removing minor contributor edges. In Figure 6, such minor edges are indicated by the dashed lines. The topological effect of removing minor edges, as shown in Figure 6, is that many pairs of binaries that had short connecting paths through minor contributors are disconnected. Our findings focused on two key aspects of the results. First, we examined the correlation between social network measures and post-release failures in the complete network and the network with minor edges removed. Second, we measured the change in the ability of a predictor to identify fault prone binaries when removing major or minor contribution edges.

SNA Metric	All Edges	Only Major No Minor
Degree	0.675	0.287
Weighted Degree	0.607	0.572
Bonacich Power	0.669	0.079
Weighted Bon. Power	0.537	0.349
Closeness	0.637	0.063
Farness	-0.636	-0.061
Reach	0.652	0.075
Betweenness	0.624	0.106

TABLE 6: Correlation of Social Network Analysis metrics on the contribution network with post-release failures. The first column is for the entire contribution network and the second column is with minor contributor edges removed. Removing the minor edges drops the correlations considerably.

Metric	All Edges	Only Major No Minor	Only Minor No Major
Precision	75.0%	43.7%	84.4%
Recall	81.7%	58.3%	87.9%
F-Score	78.2%	49.9%	86.1%
AUC	78.9%	74.3%	90.5%

TABLE 7: Performance of network based failure predictors for Vista

Table 6 shows the strength of relationship of eight network measures with post-release failures. These particular metrics are displayed because they had the highest correlation with failures. The first column shows the values for the complete network and the second contains values for the network with minor contribution edges removed. The values dropped for all metrics, some quite dramatically. These findings clearly indicate that the edges from minor contributors embody much of the important structure of the contributions graph. So much so that their removal results in a decrease in the discriminatory power of these metrics.

We also built a predictor from these measures for identifying fault prone binaries in Vista using the same approach as Pinzger *et al.* [3]. They trained a logistic regression model on a randomly chosen two thirds of the binaries in the contribution network and then evaluated the model based on its results when classifying the remaining third.

This process was repeated fifty times, each with a different random split of the data and the measures of performance, precision, recall, F-score, and area under the ROC curve (AUC) — standard measures of information retrieval [39] — were averaged across all runs.

Their original model based on the complete network identified 90% of the fault prone binaries and 85% of its fault prone predictions were correct. When the predictor was trained using the same methods on the network with minor contributors removed, it identified only 55% of the fault prone binaries and around 60% of its fault prone

Release	Metric	All Edges	Only Major No Minor	Only Minor No Major
1.5	Precision	46.2%	27.3%	66.5%
	Recall	62.8%	54.4%	69.3%
	F-Score	52.6%	35.7%	67.4%
	AUC	81.1%	76.6%	84.2%
2.0	Precision	34.7%	6.9%	26.3%
	Recall	62.0%	44.3%	59.5%
	F-Score	43.9%	12.7%	34.4%
	AUC	78.9%	72.0%	73.9%

TABLE 8: Performance of network based failure predictors for Firefox

predictions were correct. In Pinzger's formulation of the prediction approach, random guessing would result in 50% for both measures. Thus a predictor based on network measures for a network containing major contributor only does marginally better than one that chose binaries purely at random. Table 7 shows the performance when a predictor is trained on the complete network, the network with minor contributions removed and major contributions removed, as well as the performance of a random guessing approach.

We replicated this technique on the Firefox and Eclipse data and examined the performance of the defect predictor for the complete contribution networks as well as networks with major contributor edges and minor contributor edges removed. The results for our Firefox analysis are shown in Table 8. In both cases, removing the minor contribution edges decreases the performance much more than removing the major contribution edges and for the 1.5 release, the minor contribution network performed better than the model based on the complete network.

The results for our Eclipse data, shown in table Table 9, was mixed. In some of the releases (2.0, 3.0, 3.2, 3.3), the predictor based on networks with no minor contribution edges performed just as well as the complete network. In others (2.1, 3.1) and in other releases they performed far worse. The results are also inconcluside with models based on networks with minor contribution edges removed. In releases that do show a difference between the predictive power between the different networks, the difference is sometimes large and statistically significant ($p \ll 0.01$) even after adjusting for multiple hypothesis testing [40]. Thus, our data are not just a result of running a enough experiemtns that "something is bound to be significant." Until a deeper investigation is made into the differences between Eclipse releases in terms of characteristics such as development policies and practices, size and makeup of the development team, and architectural changes, we must report that our examination of the effect of different types of ownership edges on network-based failure prediction is inconclusive for the Eclipse project.

We therefore conclude that for Vista, the minor contribution edges provide the "signal" used by defect predictors that are based on the contribution network. Without them, the ability to predict failure prone components is greatly diminished, further supporting our hypothesis that they

Release	Metric	All Edges	Only Major No Minor	Only Minor No Major
2.0	Precision	72.4%	69.6%	51.4%
	Recall	75.3%	76.7%	65.3%
	F-Score	72.7%	71.8%	54.0%
	AUC	85.7%	84.8%	66.8%
2.1	Precision	66.4%	59.2%	44.6%
	Recall	70.1%	69.7%	53.2%
	F-Score	67.0%	62.4%	46.0%
	AUC	74.1%	72.8%	52.2%
3.0	Precision	73.7%	72.4%	63.2%
	Recall	77.8%	79.2%	76.0%
	F-Score	75.0%	74.9%	66.6%
	AUC	84.7%	84.3%	79.1%
3.1	Precision	53.9%	43.1%	61.8%
	Recall	64.3%	56.6%	60.4%
	F-Score	57.1%	47.1%	59.8%
	AUC	71.8%	58.6%	66.4%
3.2	Precision	60.9%	65.4%	60.3%
	Recall	76.0%	81.1%	72.7%
	F-Score	66.8%	71.6%	63.9%
	AUC	80.2%	83.0%	70.2%
3.3	Precision	54.9%	55.9%	72.9%
	Recall	69.1%	70.9%	73.8%
	F-Score	59.9%	60.9%	70.7%
	AUC	72.7%	72.8%	82.3%

TABLE 9: Performance of network based failure predictors for Eclipse

are strongly related to software quality. The results for Firefox and Eclipse are less conclusive.

8 Discussion

Our findings are valuable in a number of ways. We have shown that for Windows Vista, ownership does indeed have a relationship with code quality. This observation is an actionable result, as this is an aspect of software development that can be controlled to some degree by management decisions on development process and policies. In all projects, the addition of Minor improved the regression models for both pre and post-release failures to a statistically significant degree. Thus *hypothesis 1 is empirically supported.*

The analysis of Ownership is a little bit different. In this case, we saw a smaller, but still statistically significant effect in Vista, but the effect was only sometimes significant in Eclipse and Firefox. We conclude that *hypothesis 2 is supported only in the case of Windows Vista.*

We have also shown that the relationship between ownership levels and failures is stronger in a traditional, commercial development context than in projects with more of an open-source style development process. It appears that team size becomes more important than ownership level metrics when a project uses a more open source-style development process. At the meta-level, this provides strong evidence that results from studies of open source projects do not always generalize to settings where a different process is used. The process that is used may dictate the effect of other factors on software quality as well. Therefore,

when determining the applicability of a research result to a software project, the context of the study must be taken in account. Our data lends support to arguments made by Raymond and others that ownership matters less in open source contexts than it does in traditional commercial software development projects. Clearly, our findings should be interpreted with caution. We examine three large, mature projects that differ in process and ownership policy, but that also differ in other ways such as domain, language, and user base.

We hasten to point out that in our study we contrast only three large software projects from different domains. While the ownership policies and practices of these projects differ, we lack the sample size necessary to give strong quantitative support to our hypothesis and rule out other possibly confounding factors. Thus, *our findings support hypothesis 3, but further study on more projects is required for strong quantitative evidence.* Our findings open the door to further investigations of differences between "traditional" and "open-source" style development (though these are also broad generalities, they are first steps). Finally, we make the following recommendations regarding the development process based on our findings:

1) *Changes made by minor contributors should be reviewed with more scrutiny.* Changes made by minor contributors should be exposed to greater scrutiny than changes made by developers who are experienced with the source for a particular binary. When possible, major contributors should perform these code inspections. If a major contributor cannot perform all inspections, he or she should focus on inspecting changes by minor contributors.
2) *Potential minor contributors should communicate desired changes to developers experienced with the respective binary.* Often minor contributors to one binary are major contributors to a depending binary. Rather than making a desired change directly, these developers should contact a major contributor and communicate the desired change so that it can be made by someone who has higher levels of expertise.
3) *Components with low ownership should be given priority by QA resources.* Metrics such as MINOR and OWNERSHIP should be used in conjunction with source code based metrics to identify those binaries with a high potential for having many post-release failures. When faced with limited resources for quality-control efforts, these binaries should have priority.

These recommendations are currently being evaluated at Microsoft. We plan to investigate the relationship of the ownership measures used in this paper with software quality in other projects at Microsoft that differ in size and domain. Further, we plan to observe the results of projects that follow these recommendations.

9 Conclusion

We have examined the relationship between ownership and software quality in three projects with different development styles. We found that while high levels of ownership, specifically operationalized as high values of OWNERSHIP, and low values of MINOR, are associated with less defects, the effect is stronger in commercial settings where there are ownership practices in place than in open source settings.

An investigation into the effects of minor and major contributions on network based defect prediction found that removing minor contribution edges severely impaired predictive power for Vista and FIREFOX, but was inconclusive in ECLIPSE. We also found that in Vista, when a component has a minor contributor, the same developer is a major contributor to a dependent component approximately half of the time. Changes to policies regarding tasks that would lead to this behavior, such as bug fixing, may be useful in Vista, but less so in ECLIPSE, where this phenomenon is not as common.

For organizations where ownership has a strong relationship with defects, we have presented recommendations which are currently being evaluated at Microsoft.

References

[1] M. Cataldo, P. Wagstrom, J. Herbsleb, and K. Carley, "Identification of coordination requirements: implications for the Design of collaboration and awareness tools," *Proceedings of the 2006 20th anniversary conference on Computer supported cooperative work*, pp. 353–362, 2006.

[2] N. Nagappan, B. Murphy, and V. Basili, "The influence of organizational structure on software quality: an empirical case study," in *Proc. of the 30th international conference on Software engineering*, 2008.

[3] M. Pinzger, N. Nagappan, and B. Murphy, "Can developer-module networks predict failures?" in *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, 2008.

[4] C. Bird, N. Nagappan, P. Devanbu, H. Gall, and B. Murphy, "Putting it All Together: Using Socio-Technical Networks to Predict Failures," in *Proceedings of the 17th International Symposium on Software Reliability Engineering.* IEEE Computer Society, 2009.

[5] E. S. Raymond, *The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary.* Sebastopol, California: O'Reilly and Associates, 1999.

[6] C. Bird, N. Nagappan, P. Devanbu, H. Gall, and B. Murphy, "Does distributed development affect software quality? an empirical case study of windows vista," in *Proc. of the International Conference on Software Engineering*, 2009.

[7] E. J. Weyuker, T. J. Ostrand, and R. M. Bell, "Do too many cooks spoil the broth? using the number of developers to enhance defect prediction models," *Empirical Softw. Engg.*, vol. 13, no. 5, pp. 539–559, 2008.

[8] A. Meneely and L. A. Williams, "Secure open source collaboration: an empirical study of linus' law," in *Proceedings of the ACM Conference on Computer and Communications Security*, 2009.

[9] K. Beck and C. Andres, *Extreme Programming Explained: Embrace Change.* Addison-Wesley Reading, MA, 2005.

[10] W. Boh, S. Slaughter, and J. Espinosa, "Learning from experience in software development: A multilevel analysis," *Management Science*, vol. 53, no. 8, pp. 1315–1331, 2007.

[11] B. Curtis, H. Krasner, and N. Iscoe, "A field study of the software design process for large systems," *Communication of the ACM*, vol. 31, no. 11, pp. 1268–1287, 1988.

[12] A. Mockus and J. D. Herbsleb, "Expertise browser: a quantitative approach to identifying expertise," in *Proc. of the 24th International Conference on Software Engineering*, 2002.

[13] D. W. McDonald and M. S. Ackerman, "Expertise recommender: a flexible recommendation system and architecture," in *Proc. of the ACM conference on Computer supported cooperative work*, 2000.

[14] T. Fritz, G. Murphy, and E. Hill, "Does a programmer's activity indicate knowledge of code?" in *Proc. of the ACM SIGSOFT symposium on The foundations of software engineering*. ACM, 2007, p. 350.

[15] A. Mockus and D. Weiss, "Predicting risk of software changes," *Bell Labs Technical Journal*, vol. 5, no. 2, pp. 169–180, 2000.

[16] A. Mockus, "Succession: Measuring transfer of code and developer productivity," in *Proceedings of the 31st International Conference on Software Engineering*, 2009.

[17] E. Darr, L. Argote, and D. Epple, "The acquisition, transfer, and depreciation of knowledge in service organizations: Productivity in franchises," *Management Science*, vol. 41, no. 11, pp. 1750–1762, 1995.

[18] M. Sacks, *On-the-Job Learning in the Software Industry. Corporate Culture and the Acquisition of Knowledge.* Quorum Books, 88 Post Road West, Westport, CT 06881., 1994.

[19] R. Banker, G. Davis, and S. Slaughter, "Software development practices, software complexity, and software maintenance performance: A field study," *Management Science*, vol. 44, no. 4, pp. 433–450, 1998.

[20] P. Robillard, "The role of knowledge in software development," *Communications of the ACM*, vol. 42, no. 1, p. 92, 1999.

[21] V. Basili and G. Caldiera, "Improve Soft-ware Quality by Reusing Knowledge and Experience," *Sloan Management Review*, vol. 37, pp. 55–55, 1995.

[22] R. Kraut and L. Streeter, "Coordination in software development," *Communications of the ACM*, vol. 38, no. 3, pp. 69–81, 1995. [Online]. Available: http://portal.acm.org/citation.cfm?id=203345

[23] F. Brooks, *The Mythical Man-Month: Essays on Software Engineering, 20th Anniversary Edition.* Addison-Wesley, 1995.

[24] C. Bird, D. Pattison, R. D'Souza, V. Filkov, and P. Devanbu, "Latent social structure in open source projects," in *Proc. of the ACM SIGSOFT International Symposium on Foundations of software engineering*, 2008.

[25] V. Basili, G. Caldiera, and H. Rombach, "The Goal Question Metric Approach," *Encyclopedia of Software Engineering*, vol. 1, pp. 528–532, 1994.

[26] T. Zimmermann, N. Nagappan, H. Gall, E. Giger, and B. Murphy, "Cross-project defect prediction: a large scale experiment on data vs. domain vs. process," in *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2009.

[27] Billy Biggs (IBM Employed Eclipse Developer), Personal Interview, July 2009.

[28] A. Bachmann and A. Bernstein, "Data Retrieval, Processing and Linking for Software Process Data Analysis," University of Zurich, Department of Informatics, Tech. Rep., 2009.

[29] J. Śliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" in *Proceedings of the international workshop on Mining software repositories*, 2005.

[30] T. Ostrand, E. Weyuker, and R. Bell, "Where the bugs are," in *Proceedings of the ACM SIGSOFT international symposium on Software testing and analysis*, 2004.

[31] N. Nagappan and T. Ball, "Use of relative code churn measures to predict system defect density," *Proceedings of the 27th International Conference on Software Engineering*, pp. 284–292, May 2005.

[32] K. El Emam, S. Benlarbi, N. Goel, and S. N. Rai, "The confounding effect of class size on the validity of object-oriented metrics," *IEEE Transactions of Software Engineering*, vol. 27, no. 7, pp. 630–650, 2001.

[33] S. Dowdy, S. Wearden, and D. Chilko, *Statistics for research*, 3rd ed. John Wiley & Sons, 2004.

[34] A. Srivastava, J. Thiagarajan, and C. Schertz, "Efficient Integration Testing using Dependency Analysis," Microsoft Research, Tech. Rep. MSR-TR-2005-94, 2005.

[35] M. Molloy and B. Reed, "A critical point for random graphs with a given degree sequence," *Random Struct. Algorithms*, vol. 6, no. 2-3, pp. 161–179, 1995.

[36] M. E. J. Newman, S. H. Strogatz, and D. J. Watts, "Random graphs with arbitrary degree distributions and their applications," *Phys. Rev. E*, vol. 64, no. 2, p. 026118, Jul 2001.

[37] R. Milo, N. Kashtan, S. Itzkovitz, M. E. J. Newman, and U. Alon, "On the uniform generation of random graphs with prescribed degree sequences," *Arxiv preprint cond-mat/0312028*, 2003.

[38] S. Wasserman and K. Faust, *Social network analysis: Methods and applications.* Cambridge University Press, 1994.

[39] F. W. Lancaster, *Information Retrieval Systems: Characteristics, Testing, and Evaluation*, 2nd ed. Wiley, 1979.

[40] Y. Benjamini and Y. Hochberg, "Controlling the False Discovery Rate: A Practical and Powerful Approach to Multiple Testing," *Journal of the Royal Statistical Society. Series B (Methodological)*, vol. 57, no. 1, pp. 289–300, 1995.