Don't Touch My Code! Examining the Effects of Ownership on Software Quality

Christian Bird
Microsoft Research
cbird@microsoft.com

Nachiappan Nagappan
Microsoft Research
nachin@microsoft.com

Brendan Murphy
Microsoft Research
bmurphy@microsoft.com

Harald Gall
University of Zurich
gall@ifi.uzh.ch

Premkumar Devanbu
University of California, Davis
ptdevanbu@ucdavis.edu

ABSTRACT

Ownership is a key aspect of large-scale software development. We examine the relationship between different ownership measures and software failures in two large software projects: Windows Vista and Windows 7. 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. We also empirically identify 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.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics—*Process metrics*

General Terms

Measurement, Management, Human Factors

Keywords

Empirical Software Engineering, Ownership, Expertise, Quality

1. INTRODUCTION

Many recent studies [6, 9, 26, 29] 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 a relationship between the number of people working on a binary and failures [5, 26]. However, to our knowledge, the effect of ownership has not been studied in depth in commercial contexts. Based on our observations and discussions with project managers, 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 some aspects of software which are known to be related to defects such as 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.

We have observed that many industrial projects encourage high levels of code ownership. In this paper, we examine ownership and software quality. We make the following contributions in this paper:

1. We define and validate measures of ownership that are related to software quality.
2. We present an in depth quantitative study of the effect of these measures of ownership on pre-release and post-release defects for multiple large software projects.
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.

2. THEORY & RELATED WORK

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

Rahman & Devanbu [30] examined the effects of ownership & experience on quality in several open-source projects, using a fine-grained approach based on fix-inducing fragments of code, and report findings similar to those of our paper. However, they operationalize ownership differently,

ESEC/FSE'11, September 5–9, 2011, Szeged, Hungary.

and ownership policies and practices in OSS and commercial software are quite different. Thus the similarity of effect is striking. Furthermore, Rahman & Devanbu do not study the relationship of minor contribution on software dependencies; nor do they consider social network measures.

Weyuker *et al.* [35], examined 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. We differ in that we examine the *proportion* of contributions made by each developer to a component. Further, we are not interested in prediction, but rather determining if there is a statistically significant *relationship* between ownership and failures.

Similarly, Meneely and Williams examined the relationship of the number of developers working on parts of the Linux kernel with security vulnerabilities [19]. They found that when more than nine developers contribute to a source file, it is sixteen times more likely to include a security vulnerability.

New methods such as Extreme Programming (XP) [4] profess collective code ownership but there has been little 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 [7]. In a qualitative study of 17 commercial software projects, Curtis *et al.* [10] 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.* [22] and Expertise Recommender from McDonald and Ackerman [18] 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 [15].

Mockus and Weiss used properties of an individual change to predict the probability of that change causing a failure [23]. They found that changes made by developers that were more experienced with a piece of code were less likely to induce failure. Three of their fourteen measures capture the experience of a developer history (by counting prior changes) and they were significant in prediction.

In a study of offshoring and succession in software development [21], 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 [11]. 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 [32]. Banker *et al.* found that increased experience increases a developer's knowledge of the architectural domain of the system [1]. 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 [31]. Indeed, Basili and Caldiera present an approach for improving quality in software development through learning and experience by establishing "experience factories" [2]. 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 [16]. In fact, another of the top three problems identified in Curtis' study [10] was "communication and coordination breakdowns." Working in such a group always creates a need for sharing and integrating knowledge across all members [8]. Cataldo *et al.* showed that communication breakdowns delay tasks [9]. 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

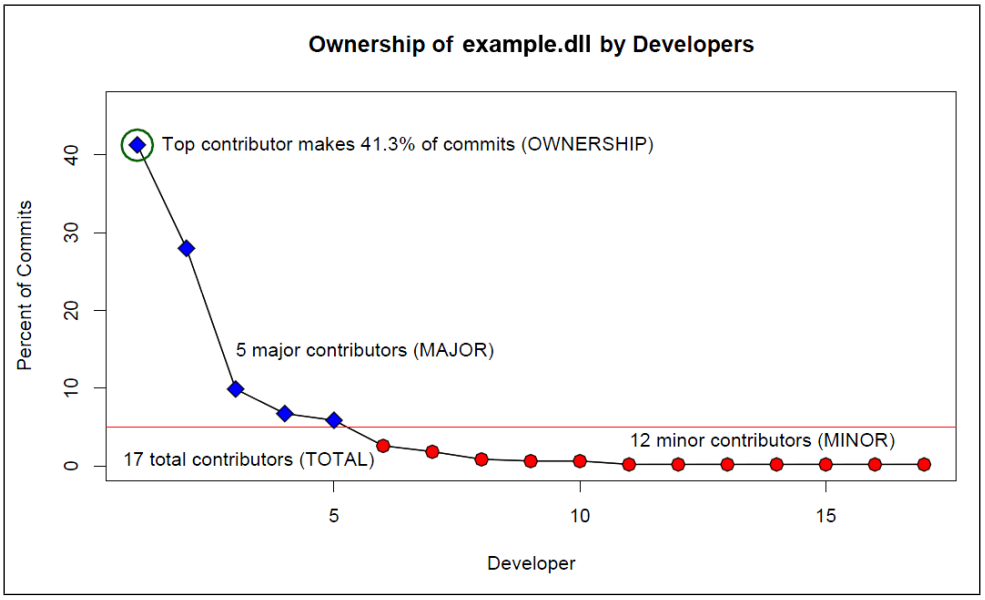


Figure 1: Graph of the proportion of commits to `example.dll` by developers during the Vista development cycle, showing the four measures of ownership used in this paper.

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.

3. TERMINOLOGY AND METRICS

We adopt Basili's goal question metric approach [3] 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:

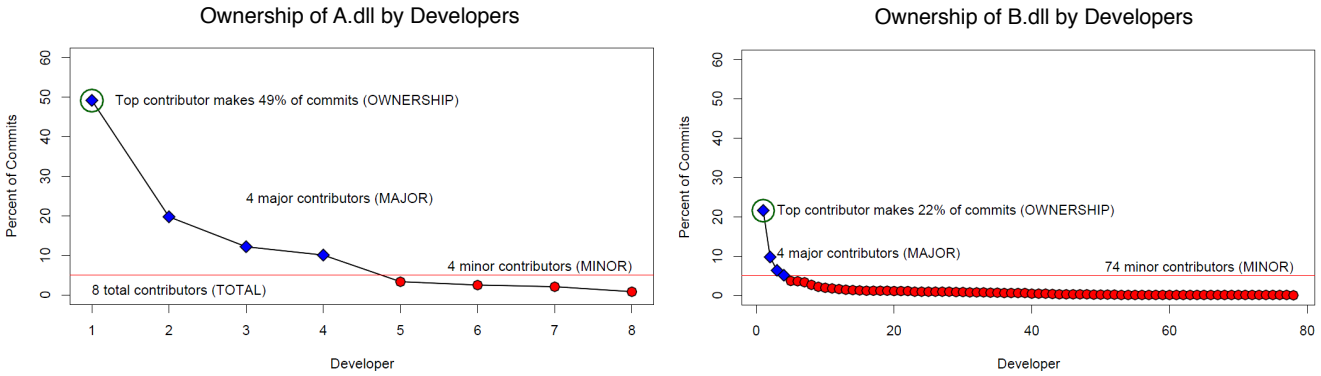
1. Are higher levels of ownership associated with less defects?
2. Is there a negative effect when a software entity is developed by many people with low ownership?
3. Are these effects related to the development process used?

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, a component is a compiled binary.
- **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 `ie9.dll` and there are a total of 100 commits to `ie9.dll` 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.

Note that we examine the number of *changes* to a component made by a developer rather than the actual number of lines modified. Within Windows, each change corresponds to one fix or enhancement and individual changes are quite small, usually on the order of tens of lines. We use number of changes because each change represents an "exposure" of the developer to the code and because the previous measure of experience used by Mockus and Weiss also used the number of changes. However, prior literature [14] has shown high

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



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

Figure 2: Ownership graphs for two binaries in Windows

correlation (above 0.9) between number of changes and number of lines contributed and we have found similar results in Windows, indicating that our results would not change significantly. 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 `example.dll` in Windows, 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 `example.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 software components that have fewer.

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.

If the number of minor contributors negatively affects software quality, the next question to ask is, "Why do some binaries have so many minor contributors?" We have observed both at Microsoft and also within OSS projects such as Python and Postgres, that during the process of maintenance, feature addition, or bug fixing, owners of one component often need to modify other components that the first relies on or is relied upon by. As a simple example, a developer tasked with fixing media playback in Internet Explorer may need to make changes to the media playback interface library even though the developer is not the designated owner and has limited experience with this component. This leads to our hypothesis.

Hypothesis 3 - Minor contributors to components will be Major contributors to other components that are related through dependency relationships

Finally, if low-expertise contributions do have a large impact on software quality, then we expect that defect prediction techniques will be affected by their inclusion or removal. We therefore replicate prior defect prediction techniques and compare results when using all data, data derived only from changes by minor contributors and, and data derived only from changes to major contributors. We expect that when data from minor contributors is removed, the quality of the defect prediction will suffer.

Hypothesis 4 - Removal of minor contribution information from defect prediction techniques will decrease performance dramatically.

Category	Metric	Windows Vista		Windows 7	
		Pre-release Failures	Post-release Failures	Pre-release Failures	Post-release Failures
Ownership Metrics	TOTAL	0.84	0.70	0.92	0.24
	MINOR	0.86	0.70	0.93	0.25
	MAJOR	0.26	0.29	-0.40	-0.14
	OWNERSHIP	-0.49	-0.49	-0.29	-0.02
"Classic" Metrics	Size	0.75	0.69	0.70	0.26
	Churn	0.72	0.69	0.71	0.26
	Complexity	0.70	0.53	0.56	0.37

Table 1: Bivariate Spearman correlation of ownership and code metrics with pre- and post-release failures in Windows Vista and Windows 7. All correlations are statistically significant except for that of OWNERSHIP and post-release failures in Windows 7.

5. DATA COLLECTION AND ANALYSIS

This data presents an opportunity to investigate hypotheses regarding code ownership. In this study, we examine Windows Vista and Windows 7.

Windows Vista and Windows 7 is developed entirely by Microsoft, who have processes and policies that favor strong code ownership. Windows Vista and 7 were developed by a few thousand software developers and is composed of thousands of individual executable files (`.exe`), shared libraries (`.dll`), and drivers (`.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 7 and include pre-release defects as well as post-release failures in Vista and 7 as software quality indicators.

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). 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. Although the source code management system uses branches heavily, we only recorded changes from developers that were edits to the source code. Branching operations (e.g. branching and merging) were not counted as changes.

We also gathered both pre-release and post-release software failures for all three projects. 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, but cannot reliably trace them to the source files that caused the failures. 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. This information is gathered from both the source code repositories and the build process.

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 (shown in Table 1). The results indicated that pre- and post-release defects in had strong relationships with MINOR, TOTAL, and OWNERSHIP. Windows 7 binaries have few post-release failures. Thus the correlation values between metrics and and post-release failures are noticeably lower than the other failure categories (although all except the correlation with OWNERSHIP are still statistically significant).

However, we also observed some relationship between code attributes and ownership metrics. For example, Figure 2 shows data for two anonymized binaries in Windows with vastly different ownership profiles. 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 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 [25, 28] 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 [13].

To overcome this problem, we used *multiple linear regression*. Linear regression, 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). We stress that while our regression analysis does use failures as the dependent variable in our models, the purpose of this paper is **not** to predict failures.

Second, linear regression enables us to examine the effect of one or more variables on an outcome when controlling for other variables. We use it for this purpose in an effort to examine the *relationship* of our ownership measures *when controlling for source code characteristics such as size, complexity, and churn*.

A 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

Model	Windows Vista		Windows 7	
	Pre-release Failures	Post-release Failures	Pre-release Failures	Post-release Failures
Base (code metrics)	26%	29%	24%	18%
Base + TOTAL	40%*(+14%)	35%*(+6%)	68%* (+35%)	21%* (+3%)
Base + MINOR	46%*(+20%)	41%*(+12%)	70%* (+46%)	21%* (+3%)
Base + MINOR + MAJOR	48%*(+2%)	43%*(+2%)	71%* (+1%)	22% (+1%)
Base + MINOR + MAJOR + OWNERSHIP	50%*(+2%)	44%*(+1%)	72%* (+1%)	22% (+0%)

Table 2: 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.

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 and over-fitting, 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 [12].

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, which led to a similar skew in the residuals. When we transformed the dependent variable to be the `log` of the number of failures, the skew diminished, and the residuals 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.

6. RESULTS

We now present the results of our analysis of Windows Vista and Windows 7. Table 2 illustrates the results of our analysis. We denote with an asterisk*, cases where a 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 model without the added variable. For example, in Table 2 the Base+MINOR +MAJOR model in Vista explains 48% of the variance in pre-release failures which is 2% more than the Base+MINOR model which explains 46%. the Base+MINOR model explains 20% more of the variance in pre-release failures than the Base model. Adding an independent variable to a model can never decrease the variance explained, so we use the adjusted R^2 measure which penalizes models that have additional variables.

We built five statistical models of failures for pre- and post-release defects in Windows Vista and Windows 7 (summarized in Table 2). 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 Vista and 24% and 28% in Windows 7.

In the second model, we added TOTAL to the classic variables. This examines the effect of team size on defects and does not include any measures of the proportion of contributions made by individual members. All models exhibitted a statistically significant improvement in variance explained.

Next, we added MINOR to the set of predictor variables in the base model. This was done to determine if the total number of developers has a different effect on quality than the number of minor contributors. 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%. The gains shown by MINOR were stronger than those shown by TOTAL for both types of failures to a statistically significant degree, in all cases except for post-release failures in Windows 7, indicating that MINOR has a larger effect on failures.

The addition of MAJOR and OWNERSHIP showed smaller gains, but were often still statistically significant. We found similar results regardless of the order that these variables were added to the models. OWNERSHIP was found to have a negative relationship with failures to a statistically significant degree and MAJOR had a positive relationship, but was much smaller than MINOR. MINOR still showed more of an effect than OWNERSHIP and MAJOR even when it was added last (not shown). The final models account for up to 72% of variance in failures. In all cases, ownership had a stronger relationship with pre-release failures than post-release failures and the models in general were less explanatory. This may indicate that there are already measures being taken (e.g. increased testing, more stringent quality controls, etc.) between implementation completion and release to counteract the effects of poor ownership.

For all metrics that measure ownership levels there is a clear trend of having a statistically significant relationship to failures in Windows. In all cases, MAJOR and OWNERSHIP show less of an effect than MINOR or TOTAL, indicating that the number of higher-expertise contributors has marginal effect on quality, although the results are still statistically significant.

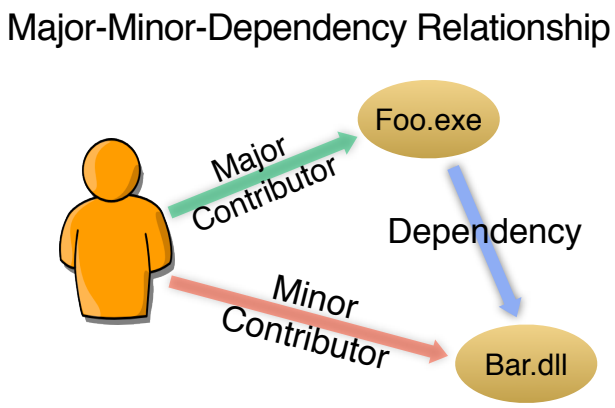


Figure 3: Illustration of the major-minor-dependency relationship commonly observed in Vista

The results of our analysis of ownership in both releases of Windows 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.*
4. *Measures of ownership and standard code measures show a much smaller relationship to post-release failures in Windows 7.*

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 both releases of Windows. Since Microsoft has the capability to make changes to practices based on these findings, we were eager to gain a deeper understanding of this phenomenon. To this end, we performed two more detailed analyses in order to examine the minor contributors further.

First, we observed that almost all developers were major contributors to some binaries and minor contributors to others; very few developers never played a major contributor role. 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 one to which she is a minor contributor?*

Second, we adapted a fault prediction study carried out by Pinzger *et al.* [29] and examined the effect of modifying the study in ways related to ownership.

7.1 Dependency Analysis

The majority of developers that contributed to Windows acted as major contributors to some binaries and minor contributors to others. There were very few 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 3.

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 [9]. We have no record of the communication between developers of Windows. 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.

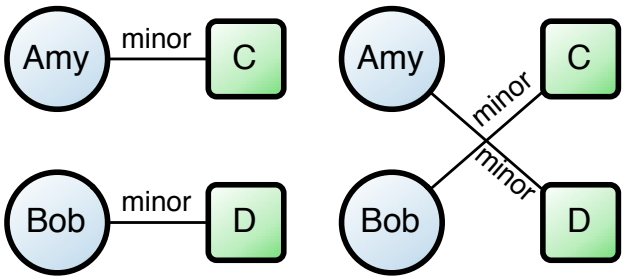
To investigate this further, we employed a static analysis tool, MaX [33], to detect dependency relationships between binaries. MaX uses debugging information files that are generated during compilation to identify these relationships, which include method calls, read and writes to the registry, IPC, COM calls, and use of types. We were unable to obtain the required debugging information files for Windows 7 and thus limit our analysis here to Vista. Using this tool, we constructed a dependency graph that includes all of the binaries in Windows 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 [20, 24, 27]. A phenomenon is judged statistically significant if the actual, observed phenomenon occurs rarely in the generated sample graphs. Following previous techniques [24, 27], we use a *graph-rewiring* method to bootstrap our random ensemble, based on the ob-

	Windows Vista				Windows 7			
	Pre-release		Post-release		Pre-release		Post-release	
SNA Metric	Minor	Major	Minor	Major	Minor	Major	Minor	Major
Degree Centrality	0.861	0.909	0.668	0.599	0.931	0.797	0.269	0.319
Closeness Centrality	0.624	-0.098	0.602	0.107	0.737	0.167	0.130	0.013
Reachability	0.647	-0.091	0.618	0.119	0.747	0.176	0.135	0.018
Betweenness Centrality	0.703	0.146	0.601	0.132	0.748	0.285	0.289	0.154
Hierarchy	0.420	-0.273	0.176	-0.244	0.298	-0.302	0.136	-0.055
Effective Size	0.775	0.311	0.649	0.286	0.884	0.391	0.223	0.196

Table 3: Correlation of Social Network Analysis metrics on the contribution network with pre- and post-release failures. Columns labeled "Minor" are correlations of failures with metrics computed on networks composed only of minor contribution edges. Columns labeled "Major" are from networks made up of major contribution edges. For the majaority of metrics, removing the minor edges drops the correlations considerably. For some metrics, the direction of correlation actually changes for "Major".



(a) before (b) after

Figure 4: 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.

served 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" for a developer has an actual dependency on a "minorly contributed component" for the same developer 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 as shown in Figure 4. 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.

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 [29]. A contribution network is composed of 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 5 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 [34]. Pinzger *et al.* found that these measures had a strong relationship with post-release failures in Windows Vista and in a prior study [6] we found that these

measures were able to predict failures in Eclipse accurately as well.

Specifically, Pinzger *et al.* 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 and major contributor edges. In Figure 5, such minor edges are indicated by the dashed lines. The topological effect of removing minor edges, as shown in Figure 5, 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.

Table 3 shows the strength of relationship of six network measures with pre- and post-release failures. These particular metrics are chosen because they had the highest correlation with failures among those collected. Columns labelled with Minor shows the correlations of SNA metrics calculated on networks that contain only minor contribution edges and those labelled with Major shows correlations from networks of major contribution edges. For all metrics except for node degree, the networks that consider major contributions have dramatically lower correlations. In fact, for the case of Hierarchy, the sign of the correlation is negative, indicating that higher value of hierarchy in the major contribution networks were associated with fewer failures. 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 Windows Vista and Windows 7 using the same approach as Pinzger *et al.* [29]. 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, and F-score — standard measures of information retrieval [17] — 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 (their evaluation was based on a prior Windows release). When the predictor was trained using the same methods on the network with minor contributors removed, it identified only 58% of the fault prone binaries and around 44% of its fault prone 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 4 shows the performance when a predictor is trained on the complete network as well as the networks with minor contributions removed and major contributions removed.

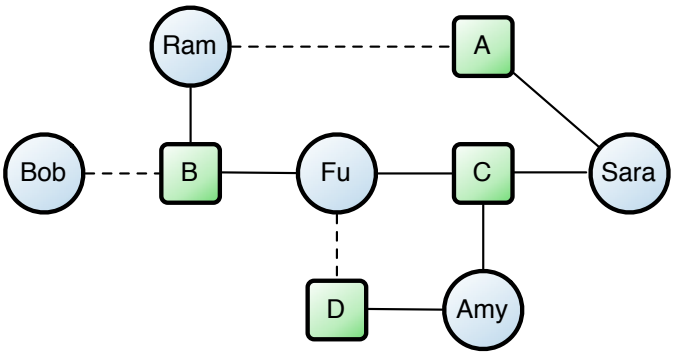


Figure 5: An example contribution network. Boxes represent binaries and circles represent developers who contributed to them. A dashed line between a binary and developer indicates a minor contributor relationship.

We also show results for pre-release failures in Vista as well as pre- and post-release failures for Windows 7. In all cases, models built on minor contributions performed better than those based on major contributions to a statistically significant degree. In the case of Vista post-release failures, minor contribution prediciton models perform better than models based on the entire network, and models based on the entire network were never statistically better than those based on minor contributions

We therefore conclude 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 are strongly related to software quality.

8. DISCUSSION

Our findings are valuable in a number of ways. We have shown that for both versions of Windows, 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 and monitored 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. After controlling for known software quality factors, binaries with more minor contributors had more pre- and post-release failures in both versions of Windows. Thus *hypothesis 1 is empirically supported in both projects.*

The analysis of Ownership is a little bit different. In this case, we saw a small, but still statistically significant effect in pre- and post-release failures for Vista and pre-release failures for Windows 7. Part of this may be attributable to a moderate relationship between the Minor and Ownership, but although Ownership was significant in all models when removing Minor, the effect was smaller. Nonetheless, in all cases, higher values for Ownership was associated with lower numbers of failures. We therefore conclude that *hypothesis 2 is supported in the case of Windows Vista and in pre-release data for Windows 7.*

The results of empirical software engineering studies do

	Windows Vista						Windows 7					
	Pre-release			Post-release			Pre-release			Post-release		
SNA Metric	All	Minor	Major	All	Minor	Major	All	Minor	Major	All	Minor	Major
Precision	90%	87%	83%	75%	84%	44%	89%	88%	80%	12%	11%	8%
Recall	91%	93%	91%	82%	88%	58%	92%	93%	87%	66%	75%	61%
F-Score	91%	89%	87%	78%	86%	50%	90%	90%	84%	21%	20%	14%

Table 4: Performance of network based failure predictors for pre- and post-release failures for Vista and Windows 7

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. Microsoft employs strong ownership practices and our results are much more likely to hold in other industrial settings where similar policies are in place. Examining the effect of ownership in contexts where ownership is not stressed as highly, such as in many open source software (OSS) projects, is an area of continued study as we attempt to understand the interaction between ownership, quality, and varying software processes.

For contexts in which strong ownership *is* practiced or where empirical studies are consistent with our own findings, 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.

It may not always be possible to follow these recommendations (for instance, in cases where too many potential contributors need changes to a component for one developer to handle), however they should be followed as much as possible within reason. 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 process domain (e.g. projects utilizing agile). 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 two large software development projects. We found that high levels of ownership, specifically operationalized as high values of Ownership and Major, and low values of Minor, are associated with less defects.

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. We also found that when a component has a minor contributor, the same developer is a major contributor to a dependent component approximately half of the time, uncovering at least one significant reason for high levels of minor contributions. Changes to policies regarding tasks that would lead to this behavior, such as defect resolution and feature implementation, should be implemented and evaluated.

For organizations where ownership has a strong relationship with defects, we have presented recommendations which are currently being evaluated at Microsoft. As our measures of ownership are cheap and lightweight, we encourage other researchers and practitioners to perform and report their findings of similar analyses so that we can build a body of knowledge regarding ownership and quality in various domains and contexts.

10. REFERENCES

[1] R. Banker, G. Davis, and S. Slaughter. Software development practices, software complexity, and software maintenance performance: A field study. *Management Science*, 44(4):433–450, 1998.

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

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

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

[5] 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.

[6] 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.

[7] W. Boh, S. Slaughter, and J. Espinosa. Learning from experience in software development: A multilevel analysis. *Management Science*, 53(8):1315–1331, 2007.

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

[9] 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*, pages 353–362, 2006.

[10] B. Curtis, H. Krasner, and N. Iscoe. A field study of the software design process for large systems. *Communication of the ACM*, 31(11):1268–1287, 1988.

[11] E. Darr, L. Argote, and D. Epple. The acquisition, transfer, and depreciation of knowledge in service organizations: Productivity in franchises. *Management Science*, 41(11):1750–1762, 1995.

[12] S. Dowdy, S. Wearden, and D. Chilko. *Statistics for research*. John Wiley & Sons, third edition, 2004.

[13] 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*, 27(7):630–650, 2001.

[14] S. Elbaum and J. Munson. Code churn: A measure for estimating the impact of code change. In *Proceedings of the International Conference on Software Maintenance*, 1998.

[15] 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*, page 350. ACM, 2007.

[16] R. Kraut and L. Streeter. Coordination in software development. *Communications of the ACM*, 38(3):69–81, 1995.

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

[18] 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.

[19] 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.

[20] 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.

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

[22] 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.

[23] A. Mockus and D. Weiss. Predicting risk of software changes. *Bell Labs Technical Journal*, 5(2):169–180, 2000.

[24] M. Molloy and B. Reed. A critical point for random graphs with a given degree sequence. *Random Struct. Algorithms*, 6(2-3):161–179, 1995.

[25] 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*, pages 284–292, May 2005.

[26] 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.

[27] M. E. J. Newman, S. H. Strogatz, and D. J. Watts. Random graphs with arbitrary degree distributions and their applications. *Phys. Rev. E*, 64(2):026118, Jul 2001.

[28] 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.

[29] 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.

[30] F. Rahman and P. Devanbu. Ownership, Experience and Defects: a fine-grained study of Authorship. In *Proceedings ICSE 2011, To appear*, 2011.

[31] P. Robillard. The role of knowledge in software development. *Communications of the ACM*, 42(1):92, 1999.

[32] 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.

[33] A. Srivastava, J. Thiagarajan, and C. Schertz. Efficient Integration Testing using Dependency Analysis. Technical Report MSR-TR-2005-94, Microsoft Research, 2005.

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

[35] 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.*, 13(5):539–559, 2008.