Ownership, Experience and Defects:

A Fine-Grained Study of Authorship

Foyzur Rahman and Premkumar Devanbu
Department of Computer Science
University of California, Davis, USA
{mfrahman,ptdevanbu}@ucdavis.edu

ABSTRACT

Recent research indicates that "people" factors such as ownership, experience, organizational structure, and geographic distribution have a big impact on software quality. Understanding these factors, and properly deploying people resources can help managers improve quality outcomes. This paper considers the impact of code ownership and developer experience on software quality. In a large project, a file might be entirely owned by a single developer, or worked on by many. Some previous research indicates that more developers working on a file might lead to more defects. Prior research considered this phenomenon at the level of modules or files, and thus does not tease apart and study the effect of contributions of different developers to each module or file. We exploit a modern version control system to *examine this issue at a fine-grained level*. Using version history, we examine contributions to code fragments that are actually repaired to fix bugs. Are these code fragments *"implicated"* in bugs the result of contributions from many? or from one? Does experience matter? What type of experience? We find that implicated code is more strongly associated with a single developer's contribution; our findings also indicate that an author's specialized experience in the target file is more important than general experience. Our findings suggest that quality control efforts could be profitably targeted at changes made by single developers with limited prior experience on that file.

Categories and Subject Descriptors

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

General Terms

Experimentation; Measurement; Verification

Keywords

Ownership; Experience; Collaboration; Software Quality

ICSE'11, May 21–28, 2011, Waikiki, Honolulu, HI, USA

1. INTRODUCTION

Software defects cost the US economy billions each year [1]. For several decades now, studies of the causes of defects have led to both prediction models and "actionable" approaches to change processes to improve quality. Of late, there has been increasing attention to human factors (such as organizational structures [33], code ownership [34], and developer experience [13]) as an influence on software quality.

Code ownership can be a good thing. Strong code ownership might give rise to pride of workmanship, and thus yield higher-quality results. In industry, code ownership makes it easier for managers to hold the right people accountable, and also can help find experts to whom tasks may be delegated, and questions addressed. On the other hand, excessive levels of ownership might lead to "in-bred" code whose quality suffers from a lack of wider scrutiny. *Experience* is a related, but distinct phenomenon. As a developer contributes more to a project, her experience increases, thereby it is likely that the code she writes would be of higher quality and thereby less defect prone. On the other hand, an experienced developer might be overconfident in her approach to unfamiliar code, and thus inadvertently lower its quality.

Researchers have previously studied [11, 13, 16, 30, 46] the effects of ownership and experience; but this research largely considers ownership and experience at the level of files and modules. However, several developers may contribute to each file or module; thus the effects of many developers' varying ownership and experience, on quality, can be conflated together, and give anomalous, noisy, and/or confusing results, arising from the ecological fallacy [22]. We adopt a much more precise, fine-grained approach.

Our approach is to zero-in on the lines of code that were actually changed or deleted to fix bugs. We refer to these lines as *"implicated code*[1]*"*. Using the provenance facilities (based on modern, more precise differencing algorithms) afforded by the Git version control system, we can track down the origin and authorship of individual lines of code, and thus more precisely identify the authors of implicated code. We can then contrast the experience and ownership characteristics of implicated code with "normal" (randomly chosen background) code, as we describe in detail below.

In this paper, we study the relationship of ownership and experience (at a fine-grained level) on the quality of code. We make the following contributions.

1. We find that stronger ownership by a single author is associated with implicated code.

[1]We use *"implicated"* in the sense described in WordNet, `http://wordnetweb.princeton.edu/perl/webwn?s=implicated`

2. We find lack of specialized experience on a particular file is associated with implicated code in that file.

3. We find that lack of general (non-specialized) experience is not consistently associated with implicated code.

4. *Methodology:* For our study, we use a novel, fine-grained, sampling method. We use the `blame` feature of Git to track the provenance of code fragments implicated in bugs, and compare the provenance of this "implicated" code with *randomly chosen* "unimplicated" fragments of the same size, drawn from the same file, and same point in the system's history. We argue that this approach is general, and can be used to study a range of questions as to why certain code fragments are involved in bugs, and others are not; the resulting knowledge can drive new approaches to quality control and defect detection.

These results are *actionable*: at the simplest level, our results indicate that *specialized experience* matters more than general experience: rather than throwing a random developer, at crunch time, to fix problems in an unfamiliar part of a system, we should use someone with experience on that part of the system. We discuss further implications in Section 7.

2. BACKGROUND & THEORY

Ownership: Large software projects must divide the work between members of the development team. The question of *ownership* arises here: should each member of the team own their piece of the system? Or should everyone collectively own everything? This is controversial. The Agile movement [8] argues for collective ownership, as does Raymond [37], who claims that increasing the number of collaborators will accelerate defect diagnosis. Others counter with evidence suggesting that *"too many cooks"* working on the software leads to unfocused, defect-prone contributions [34]. Also, communication and coordination overheads worsen as the team size increases [14, 17]. Researchers have also found that these overheads can slow down development [15, 20, 26] and increase defects [18].

Seifert *et al.* [41] found that an increasing number of distinct authors making change to a file may lead to more defects. Meneely *et al.* [30] used social network based measures to analyze the effect of focus and ownership on security related errors. They found that contributions by less-focused developers were associated with more security-related errors. They also found that when more than nine developers contribute to a source file, it is sixteen times more likely to include a security vulnerability. Bird *et al.* [11] find evidence that small contributions by minor developers are associated with defects, specially in more commercially-oriented development processes. There are, however, some contradicting results: Weyuker *et al.* [46] and Graves *et al.* [25] both found that the number of contributors didn't necessarily relate to fault-proneness.

The findings reported above are generally at the file or module level. However, code authorship happens at a much more fine-grained level. Indeed, different bits of code in the same file may have been contributed by different authors; version control systems can track authorship at the line level. Studies of ownership effects at a file or module level aggregate the effects of individual contributors together, thus risking the "ecological fallacy" [22] of transferring findings from the aggregate to the constituent element; there are both internal and construct validity issues with this type of ecological inference. With the fine-grained provenance information available from `git blame`, we address the question, is ownership at a fine-grained level indicative of defect-proneness? If one believes the "many eyeballs" theory of Raymond [37], shouldn't fragments of code with multiple authors be less likely to have defects?

RQ1: Is implicated code less likely to involve contributions from multiple authors?

Experience: A related, but distinct issue is *experience*. Employee experience and its impact on productivity and quality of output has been the subject of extensive study in many fields and several industries [4, 5, 35, 45]. In manufacturing, when a worker performs a task repetitively, her efficiency improves and quality of output increases [19]. Similar findings were reported for service industries [35]. Unlike manufacturing or service industries, software development is not particularly repetitive, and requires significant creativity; however, studies indicate that a developer's knowledge of the system increases as she works more on various components of the system [7, 39]. Knowledge gained from experience matters: it can lead to better ability to answer questions about previous work [23], and better quality work [31]. These findings are also supported by Robillard [38], who suggests that lack of such knowledge could hurt software quality. We contribute to this literature in two ways. First, we pursue a fine-grained approach, considering the contributions of each developer within a file, and studying the relationship of developer experience to quality at the level of these fragmentary contributions. Second, we conceptualize two distinct types of experience that can affect the quality of a developer's work: *specialized experience* in a file under consideration, and *general experience* in the entire project. We define precise measures later, but intuitively, we view general experience of an individual as a measure of the total contributions of that individual to a project, and specialized experience of an individual with respect to a file as a measure of the number of contributions of that individual to that file.

The distinction between specialized and general experience matters. Given an urgent task, a manager might have difficulty finding a person with specialized experience relevant to a given task, and so might be tempted to deputize a developer with a lot of general (not necessarily relevant) experience. Is this a temptation to which one should yield? Boh *et al.* [13] studied the impact of such specialized and general experience on developer productivity. Their findings indicate that specialized experience has much larger impact on developer *productivity* than diverse experience in unrelated systems. We consider the impact of general and specialized experience on *quality*. While Mockus and Weiss [31] considered the quality effect of both types of experience, they study the quality effect of experience at the relatively coarser granularity of a "maintenance request" (MR); a single MR may involve changes to multiple files by multiple authors, and [31] conflates the specialized & general experience of all contributing authors together using a geometric mean. We study the quality effect of experience at a finer granularity, considering the authorship of lines of code within each file, and comparing the experience of authors of lines implicated for bugs, with that of "background code" (randomly selected code with similar properties) authors. We consider specialized experience (RQ2) and general experience (RQ3) separately.

In particular, when a developer authors some lines in a file, one can reasonably assume that her prior experience on that selfsame file is the most relevant experience. We hypothesize that the more a developer contributes to a file, the less likely it is that a change authored in that file by him/her would be implicated later.

RQ2: Is a developer with higher file level authorship less likely to be associated with implicated code?

Next, we consider general experience. The more work a developer has done on *every file* of the system, perhaps the less likely they are to make mistakes. Paradoxically, prior work suggests that experienced developers are more defect prone (e.g. Erich Gamma was identified as second most defect prone programmer in Eclipse). Zeller makes the argument that this is because the most experienced developers work on the most often exercised, most complex part of the system [47]. We examine this issue at a fine-grained level, by measuring the level of experience of the authors associated with implicated code:

RQ3: Is general experience more likely to be associated with implicated code?

The above questions consider defect-proneness without addressing the type of defects. However, in most projects faults are associated with a criticality factor, *e.g.,* Apache etc. uses a notion of *Severity*. Unfortunately, fault-proneness generally considered in the research (with few exceptions [36, 42, 48]) without attention to severity. However, when considering the effect of experience on fault-proneness, one can naturally believe that experienced developers are more likely to be working on critical code, and thus more likely to be associated with critical faults. Thus when we inspect the authorship of implicated code, we might expect that implicated code associated with severe defects is more likely to be associated with senior developers, *viz.* developers with more general experience.

We therefore examine whether experienced developers are writing more severe bugs.

RQ4: Is general experience associated with implicated code for more severe defects?

Overview of Methodology Existing research addresses ownership and experience questions mostly at file or module level. We instead try to study the ownership and experience in the context of implicated code. We identify lines of code that are associated with changes that are identified as defect repairs. This code was originally called "fix-inducing code" by Sliwerski *et al.* [43] (also see [29]), as it is the code that needed repair; for example if the code line `strcpy (str2,str1);` was changed to `strncpy(str2,str1,n);` then the original line is considered fix-inducing code. We have found that the term "fix-inducing code" causes confusion, we prefer the term *"implicated code"*:

> *Implicated code* is code that is modified to fix a defect.

Figure 1 (adapted from [43]) shows how we find implicated code. We start with data linking a given bug to the revision where that bug was fixed. If a bug fix is linked to revision $n+1$, then the immediately preceding revision (*viz.,* n) should contain the relevant "unfixed" code. A `diff` between revision n and $n+1$ for each of the files that was changed in revision $n+1$ gives us the potential buggy code. We call these lines the *implicated code*. We then use the `git blame` command on all such implicated code; this produces provenance annotations (author, date, revision number where they were last changed) on each line (we note here that `git` uses a sophisticated differencing algorithm that is sensitive to whitespace changes, line breaking, and even (to some extent) code movement). In the figure, versions i, j and k contributed lines that were changed from version n to $n+1$. Versions i and j occurred before bug b was reported; version k occurred after. Following [43] we consider code attributed to version k as innocent, and not part of the implicated code associated with bug b.

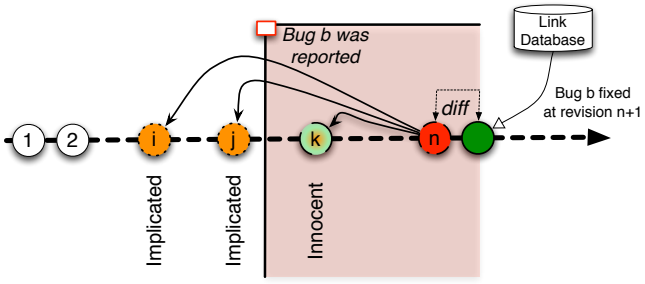

Figure 1: Finding Implicated Code

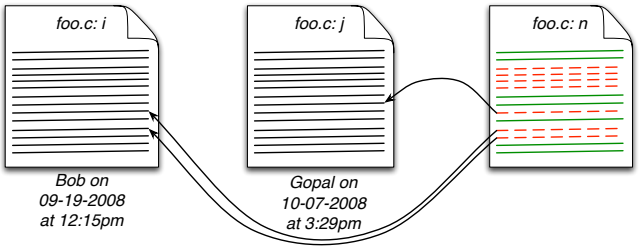

Figure 2: Implicated Code author information

Figure 2 shows how we find the author and modification time. In the figure, three of the seven implicated code lines were attributed to revision i and j which were originally committed by Bob, on 09-19-2008, and Gopal, at 10-07-2008. These details can be used to study the ownership and experience in implicated code, when compared to the rest of the code. Does the implicated code itself involve more authors? Are there any significant differences in experience level associated with implicated code?

To answer these questions, we draw upon the existing literature on ownership and experience. Girba *et al.* [24] visualize a project's evolution based on ownership changes. They define a developer's ownership of a file as the percentage of lines attributed to him/her in a file. The overall owner is the author with the highest ownership. Mockus *et al.* [32] report that developer experience is negatively correlated with failure proneness. They measure experience as the number of changes that a developer made to a system upto a given time. We define *general experience* as the cumulative developer contribution to the entire system at a given time. Our study of implicated code goes further, to consider the influence of *specialized experience*, as discussed below.

3. EXPERIMENTAL FRAMEWORK

In this section we will define all the terminologies and background of our experiment.

3.1 Revision

Source code management systems (SCM) provide a rich version history of software projects. This information includes the full history of commits to each file: timestamps, authorship, change content, and the commit log. In our study we identify each of these commits as a revision, where a revision consists of an author, a timestamp and a set of files changed in that revision. We chose GIT as our version control system, thanks to its excellent provenance facilities.

3.2 Bug-fixing Revision

Typically bugs are discovered and reported to an issue tracking system such as Bugzilla and later on fixed by the developers. Each bug report records the opening date, the fixing date, a free text bug decription, and the final, triaged Bugzilla severity. A typical bug severity in Bugzilla may have one of the following values: "blocker", "critical", "major", "normal", "minor", "trivial", "enhancement". We consider any change associated with a report in the Bugzilla database that is not labeled as an "enhancement" to be a bug.

Our study begins with links between Bugzilla bugs and the specific revision that fixes the bug—-we call this a *bug-fixing revision*. Our data is derived using several different heuristics. Various key words such as "bug", "fixed" etc. in the SCM commit log are used to flag bug-fixing revisions [2]. Also, numerical bug ids mentioned in the commit log, are linked back to the issue tracking system's identifiers [21, 44]. Then the data is crosschecked with the issue tracking system to see whether such issue identifier exists and whether its status changes after the bug-fixing commit. Finally manual inspection is used to remove spurious linking as much as possible. Each of the remaining *linked bugs* can be associated with a particular bug-fixing revision. We gratefully acknowledge the direct use of linked bugs data derived by Bachmann [6].

These bug-fixing revisions change (or delete) certain lines in the source base in order to fix bugs. These changed or deleted lines are the implicated code lines, which are the central focus of our research. New lines may also be added in the bug fixing revision, but we do not consider these "implicated", since they are part of the treatment, not the symptom.

Implicated code for a single bug may not be contiguous, in fact it may occur as several spatially separated groups of contiguous lines. Each group of contiguous lines in the implicated code is referred to as a *hunk*.

3.3 Implicated Author

For each line of implicated code, we seek to identify the author of that line of code. We use the `blame` functionality of GIT to identify the revision, time, author that introduced the line. GIT has a very accurate `blame` facility [12], which properly handles whitespace changes, and tracks code copying and movement within and even between files[2]. We then identify, at the resolution of individual lines, all the revisions, and revision authors contributing to the implicated code.

3.4 Authorship, Ownership

Based on our partitions for implicated code, we compute the proportion of contribution for each author. So, if N^b lines are changed to fix a bug b, and there are a total of m distinct authors, and the number of lines contributed by author a to the implicated code of bug b is N_a^b, then contribution ratio for author a is $r_a = \frac{N_a^b}{N^b}$. We call this *authorship* for author a. We also determine the author with the highest authorship, i.e. highest contribution ratio, and flag this author as the primary implicated code contributor for this bug.

In this paper, we use the term *ownership* of a code fragment to refer to the authorship of the highest contributor. Thus, implicated code ownership would give us the authorship of the highest contributor to that implicated code. In a similar fashion, we determine ownership of a file by considering highest contributor of a file instead of a bug.

3.5 Background Code

When making claims about properties of implicated code (such as number of contributors, experience of owner, etc), it is important to have a reference point. For example, if we claim that implicated code has more authors: the question arises, "when compared to what"? Clearly, one needs to compare implicated code with some non implicated code, to see if there is something different about the implicated code. Our approach is to compare each hunk of implicated code to a randomly chosen hunk of non-implicated code. We call these hunks chosen for comparison *background code*.

Obviously, the background code chosen for comparison to a given hunk of implicated code, should be selected both carefully and fairly. For example, we should choose hunks of the same size. A background code hunk smaller than a corresponding implicated hunk will tend to have fewer contributors than a larger hunk. Furthermore, newer code might have fewer authors than older code. Therefore, we compare each hunk of implicated code to a randomly chosen hunk of the same length, from the same file, from the same revision as the implicated code was considered. Furthermore, comments and blank lines can be expected to have different patterns of authorship and evolution than non-commentary code. For this reason, we disregard comments and blank lines in both implicated code and background code.

We use the following procedure to choose background hunks. First, we pick *locationally similar* background code, from the same file of its corresponding implicated code. Next, we choose *temporally similar* background hunks, by choosing a hunk from the same revision that the corresponding implicated code was changed to fix the bug. Next, we ensure *length similarity* by picking random contiguous hunks [28] of the same length as each such hunk in the implicated code. During the selection process, we ensure *structural integrity* by picking code hunks that do not cross function boundaries[3]. That is, we do not pick code hunks which may span multiple functions. Our algorithm tries to find background code that doesn't overlap with the corresponding implicated code. However, it may occasionally return overlapping background code when it couldn't find a non-overlapping one. Following [43] (also explained in section 2), we discard any implicated code lines that were introduced after the bug was reported. To ensure spatial and structural similarity of background code in such cases, we run a post processing on the selected background code hunks. The post processing discards i-th line of h-th hunk of background code, if the corresponding line is discarded from the implicated code (i.e. i-th line of h-th hunk of implicated code was changed after the bug was reported and was identified as innocent for that reason).

[2]See `http://www.kernel.org/pub/software/scm/git/docs/git-blame.html`, last checked July 25, 2010

[3] We use UNDERSTAND from SCITOOLS^{TM} to find function boundaries

Thus, we choose background code hunks *matched as closely as possible to the implicated code*. Since the background code hunks come from the same time, from the same file, and are of the same length as the implicated hunks, we have a greater chance of discerning any significant dissimilarities arising in the ownership level of the implicated hunks, and the experience of the authors contributing to them.

This methodology of comparing implicated code with carefully chosen background code is an important contribution of this paper, and we argue that *this is a novel experimental technique that has applications beyond this work*. It can allow study of a range of questions concerning fine-grained phenomena that can give rise to defects—such as language features, APIs, programming styles, static properties, and so on.

3.6 Experience

To measure the *general experience* of an author, we use the total number of deltas committed to the source code repository by an author up to a particular point in time. E.g. if she committed a total of 500 lines of deltas until January 1st, 2002, her general experience would be 500 at that point, but if we look at January 1st, 2010, she might have contributed 10000 lines by that time, thereby changing her general experience to 10000. We derive general experience for implicated and background hunks using the weighted geometric mean of the experience of all contributing authors, as described by Mockus [32]. In our case, weight is the proportion of either the implicated code or the background code written by a person and experience is her general experience at that point in time.

We also use a developer's authorship of a file at a point in time to measure her *specialized experience*.

4. THE DATA SETS

We chose 4 different medium- to large-sized open-source projects for our study. All have long development history, but hail from different domains. *Apache Httpd* is a widely used open source web server. *Nautilus* is the default file manager for the Gnome desktop. *Evolution* is the default email client for the Gnome desktop with support for integrated mail, address book and calender functionality. *Gimp* is a popular open source image manipulation program. All of our projects are written in C. We converted the Apache subversion repository to GIT and used the other projects' GIT repositories directly.

Name	Max. Size (LOC)	Linked bug count	Implicated hunk count	Author count
Apache	208388	478	1687	96
Evolution	531342	1022	9836	685
Gimp	947073	1601	31184	400
Nautilus	366894	534	3025	483

Table 1: Summary of study subjects

These projects represent a diverse set of application domains: a server, a client, a file browser, and a GUI application. All are non-trivial. A summary of descriptive statistics of the projects studied is presented in table 1. They range in size from 200k lines to about a million lines. The table presents details about the number of LOC, the total number of linked bugs (over the entire period), total number of implicated code fragments and number of distinct authors.

We determine all linked bugs and their associated bug-fixing revision for all the subject projects. The corresponding implicated code is found by using diff between a bug-fixing revision and its immediate preceding revision. Such implicated code may consist of multiple hunks and for each hunk we used our careful background-code-finding procedures to select similar, random non-implicated code. We also experimented with several alternative, less stringent methods to select background code. Our alternative background code selection procedure includes: relaxing function boundary requirement, keeping comments, picking one contiguous code section for all the composing hunks of a bug etc. In all those cases we found results similar to our stringent background selection procedure.

After collecting all implicated code and the corresponding background code, we use GIT BLAME on each line of the implicated code and background code to identify its *last* modifying author and the modification time. We then proceed to compare their properties to answer various research questions concerning the impact of code ownership and developer experience on software defect proneness.

5. RESULTS

We now present our results, broken down by the research questions presented earlier in Section 2. In some cases, a single research question was subject to detailed investigation using a few different measures, and we present them as **RQ1a, RQ1b**, *etc.* For illustrative purposes, we present the boxplot charts only for Apache and Evolution. Data from other projects is discussed in the text.

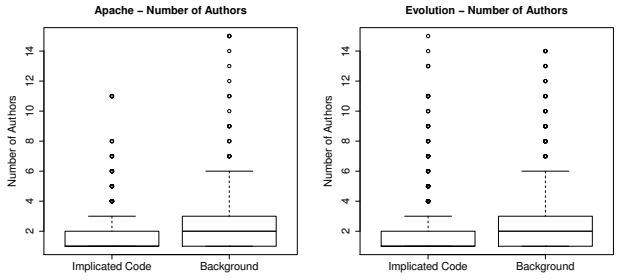

Figure 3: Number of distinct authors in implicated code and background for (a) Apache (b) Evolution.

RQ1a: *Is implicated code less likely to involve contributions from multiple authors?*

To evaluate this "more eyeballs ⇒ better code" theory, we compare the number of authors in the implicated code against the number of authors in comparable background code. Figure 3 shows the boxplot of number of authors in implicated code and background code. The boxplot suggests that the number of author is actually higher in background than implicated code. A Wilcoxon paired test (alternative hypothesis set to "number of author is greater in background code") are shown in table 2. All the p-values for RQ1a in table 2 are very low, rejecting the null hypothesis conclusively.

Despite the evidence to reject the null hypothesis in this case, if we look at the boxplots, we can notice both small range and strong positive skew, due to the limited range of number of authors (mostly 1, 2, or 3) in implicated code and comparable background code: most implicated code hunks are small, they are largely single authored. As a more sensitive test, we consider the *primary ownership* of

Name	RQ1a	RQ1b	RQ1c	RQ2a	RQ2b	RQ3	RQ4
Apache	$p \ll 0.001$	$p \ll 0.001$	0.625	0.275	0.0009	0.159	0.361
Evolution	$p \ll 0.001$	$p \ll 0.001$	0.002	$p \ll 0.001$	$p \ll 0.001$	0.999	0.304
Gimp	$p \ll 0.001$	$p \ll 0.001$	0.004	$p \ll 0.001$	$p \ll 0.001$	$p \ll 0.001$	0.076
Nautilus	$p \ll 0.001$	$p \ll 0.001$	0.002	0.0008	$p \ll 0.001$	0.0002	0.632

Table 2: The Alternative hypothesis is that the answers to the research question is "yes" in all cases. Wilcoxon test p-values for RQ1, RQ2, RQ3 and RQ4. All p-values have been adjusted using Benjamini-Hochberg procedure. p-values are the probability of the sample if the null hypothesis were true. Except RQ4, all are *paired* Wilcoxon test.

implicated code and background code, viz., the proportion of the hunks contributed by the largest contributor, thus giving us range of values upto 1.0.

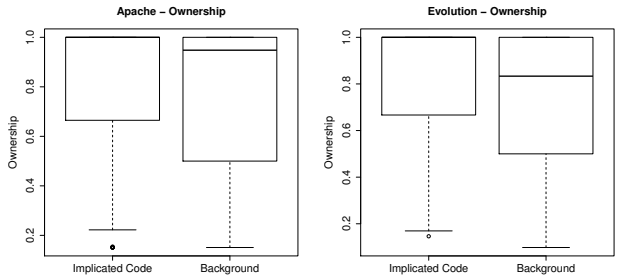

Figure 4: Ownership in implicated code and Background for (a) Apache (b) Evolution.

RQ1b: *Does implicated code have higher primary ownership?*

Figure 4 shows the boxplot of primary ownership in implicated code and background code. The boxplots are consistent with RQ1a boxplots and the p-values are shown in table 2. All the p values are again very low, thereby we reject the null hypothesis. This result provides additional evidence indicating that the implicated code tends to more frequently originate from fewer authors, when compared to similarly chosen background code.

Noticing that many of the implicated code fragments had only one single author, (and the resulting negative skew in the data) we decided to perform one bit of sensitivity analysis on the data. We discarded all singly-authored implicated code fragments, and considered just the ownership of the remaining implicated code. It's important to notice that this is a particularly stringent test: by omitting this data from implicated code, we are reducing the likelihood of detecting a difference between implicated code and background.

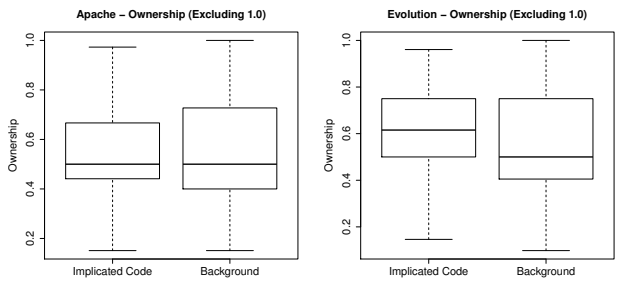

Figure 5: Ownership after discarding single author implicated code and their corresponding background code for (a) Apache (b) Evolution.

RQ1c: *Does implicated code with more than one author demonstrate higher ownership?*

We discard all the implicated code that has a single author; for fairness, we also discard the background code chosen to match this implicated code. It's important to note that it is only the background code chosen to match this buggy code that is discarded. We then compare the ownership of the remaining implicated code and background code. The boxplot of our comparison is shown in figure 5. Three out of four projects (except Apache) still give us consistent findings with RQ1a and RQ1b. The p-values are shown in table 2. This is a conservative analysis, as we selectively discarded only the singly-authored implicated code (and its corresponding background code), without selectively discarding the singly-authored background code (we of course discarded the singly-authored background code if that corresponds to a discarded implicated code). The Apache contains relatively more singly-authored implicated code (63% of linked bugs have singly-authored implicated code, while Evolution, Gimp and Nautilus have 57%, 46%, 55% respectively), and so a relatively large number of multi-authored background code got discarded in the process.

Our results from RQ1a, RQ1b and RQ1c all shows evidence of fewer authors and corresponding higher ownership in implicated code.

Implicated code is less likely to involve contributions from multiple developers.

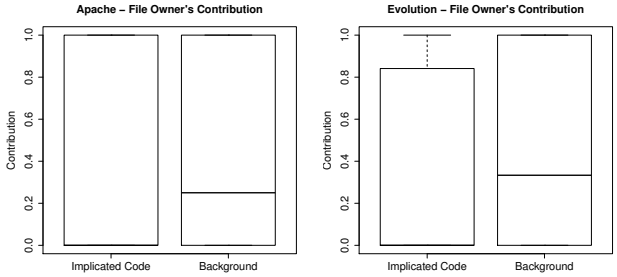

Figure 6: File owner's contribution in implicated code and background code for (a) Apache (b) Evolution.

RQ2a: *Is higher authorship by the file owner less likely to be associated with implicated code?*

One would expect that a developer's expertise on a file's code would increase significantly as she writes more of the code in the file. Does this expertise translate into better quality code? If so, we would expect to see a file's owner writing less implicated code.

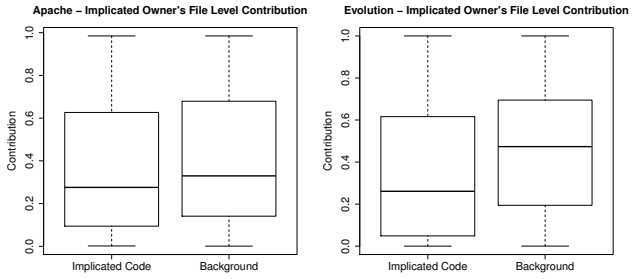

Figure 7: Implicated and background code owner's contribution at file level for (a) Apache (b) Evolution.

We define the file's owner (at the time the bug fix was performed) to be the developer with the most contributions to the file. We then compare a file's owner's contribution to the implicated code against background code. Figure 6 compares file owners' contribution in implicated code and background code. If the file owner writes less implicated code in that file, then her contribution in implicated code of that file would be less than her contribution in background code of that file (alternative hypothesis). In 3 of 4 cases (see table 2, RQ2a column) we can reject the null hypothesis, and find support for the alternative hypothesis that file owners indeed write significantly less implicated code in that file. Figure 6 shows how RQ2a is supported by the Evolution dataset, but not the Apache dataset. However, left and right side boxplots in each dataset in the figure are skewed, and do look rather similar. This is because file owners *ipso facto* write most of the code in the files they own, including both implicated code and background code. We therefore consider a more sensitive measure, by reversing the sense of ownership, and asking how much the primary contributor of the implicated code (or corresponding random fragment) contributes to the file.

RQ2b: *Do implicated code owners have lower contribution at file level?*

In RQ2a we found, in 3 of 4 cases, that file owners are less likely to introduce implicated code. We intend to investigate this trend further by asking the converse question. If the RQ2a trend holds, i.e. specialized experience in a file is associated with writing less implicated code in the same file, then we would see that the implicated code owners would have lower contribution to the file.

To investigate this, we compare the contributions of implicated code owners and background code owners to the containing file. If prior experience on the file matters, then implicated code owners should tend to have less ownership at the file level. Figure 7 presents the boxplots comparing these two samples. The figure indicates that implicated code owners have lower file level contribution than background code owners. The p-values are shown in table 2 with alternative hypothesis set to "implicated code owners have lower file level contribution than background code owners". All the p-values are highly significant, thereby rejecting the null hypothesis conclusively. The findings from RQ2a and RQ2b are largely consistent and suggest that specialized experience plays an important role to write less defective code.

Implicated code owner has *lower* contribution at file level.

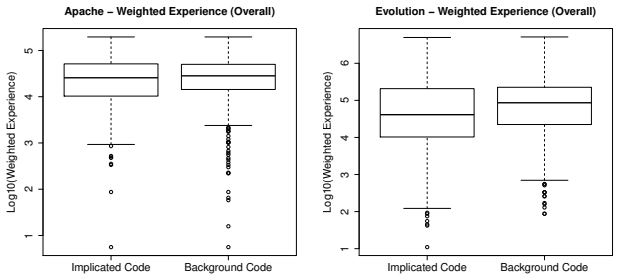

Figure 8: Weighted author experience in implicated code and background code for (a) Apache (b) Evolution.

RQ3: *Is general experience more likely to be associated with implicated code?*

Next we examine whether general experience is associated with implicated code. We compare general experience of implicated code against the general experience of background code. We measure general experience of a developer as the number of deltas she has contributed until a point in time. The weighted experience of all the hunks of the implicated code of a bug is computed by multiplying each contributing developer's experience by the number of lines she contributes to those hunks. Following Mockus [3] we use the geometric mean of these weighted values; geometric means are used because the distribution of experience is very right-skewed.

In figure 8, we compare general experience of implicated code and background code. From boxplots we observe a lower weighted experience of implicated code only for Evolution. Also, for Apache, the difference is almost indistinguishable. For other two projects (Gimp and Nautilus) we observe a higher weighted experience for implicated code. Table 2 presents p-values for Wilcoxon rank sum test with alternative hypothesis set to "Implicated code associated with more general experience than background code". We reject the null hypothesis for the Gimp and Nautilus while fail to do so for Apache and Evolution. Further study is clearly needed to shed light on this question.

General experience of author has no clear association with implicated code

RQ4: *Is general experience associated with implicated code for more severe defects?*

If experienced authors always take the brunt of programming the most risky parts of the system, it can happen, oddly enough, that they are responsible for more implicated code [47]. We come at this question somewhat indirectly, asking if experienced developers are more responsible for implicated code corresponding to severe bugs. We therefore compare weighted experience as is calculated in RQ3 but for different bug severities. Thus, we compare weighted experience of implicated code after breaking the data down by severity. We use two partitions. The *severe* partition contains "blocker" and "critical" and rest of the categories are partitioned as *not severe*. We then compare these two

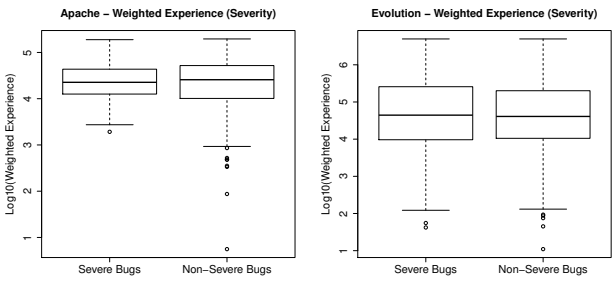

Figure 9: Weighted Author Experience for Severe Bugs and Other Bugs in (a) Apache (b) Evolution.

distributions' weighted experience using boxplots to see if the general experience is more associated with implicated code for more severe bugs (alternative hypothesis).

Figure 9 depicts the boxplot of weighted experience of corresponding implicated code for severe and non severe bugs. As is apparent from the figure and supported by the Wilcoxon rank test (p-values are presented in table 2) experienced authors may not be introducing significantly more severe defects than non experienced developers. This suggests a lack of association of experience and the authorship of code involving severe defects.

Experienced developers are not clearly associated with code implicated in severe defects.

6. DISCUSSION

Our findings have several important implications on existing software development practice. We found that implicated code is mostly contributed by a single author. This finding provides empirical support for Linus' Law (as asserted by Raymond [37]). Perhaps, at this fine-grained level, programmers could efficiently review any contributions by their peers, and perhaps serendipitously find and fix errors before they are exposed and reported into the issue tracking system.

Interestingly, this *appears to conflict* with the findings by [11, 41] which suggest that more contributors lead to more defects However, it should be noted these 2 results were both based on aggregated studies at the module and file levels, whereas ours is a fine-grained study at the line level of the implicated code. Perhaps at a coarse-grained level, it's more likely that contributors aren't as aware of each other's work; this mutual "long-distance" ignorance could lead to cross-purposes and thus errors. However, Weyuker *et al.* [46] and Graves *et al.* [25], in another coarse-grained study, found that number of contributors were not a factor in defect-proneness. Clearly, the number of authors plays a complex role in defect-proneness, and further study is needed. We speculate *e.g.,* that a code fragment with multiple authors in its backward slice is more likely to show defects if authors' contributions are pairwise distant in the same slice.

We also found that specialized experience consistently helps in writing less defective code. This suggests a very targeted inspection approach, based on prior history of the contributions to a given file; rather than directed inspection efforts at a coarse level to an entire file, this approach would direct inspectors to a *specific set* of changed or new lines, *viz.,* those written by developers into a file, when those developers had less prior experience in that file. Given the isolated nature of implicated code, written by mostly one developer, even a review of only newly introduced code may yield fewer defects. Moreover, instead of making changes themselves, less familiar developers might be required to collaborate with more familiar developers, subject to resource availability.

On the other hand, the importance of general experience on implicated code is not clearly evident. More research is needed to assess the precise impact. But the findings of RQ3 and RQ4 suggest that general experience may not be very effective to ensure code quality. However, it is not our intention to downplay the value of general experience. Earlier findings suggest that general experience could cut down integration time and foster productivity [13]. Also, experience of one domain may provide an analogous solution for another domain [40]. Moreover, repetitive use of particular APIs or working on a particular system could create *episodic knowledge*. Such episodic knowledge could help in writing succinct code or designing a better software [38].

7. THREATS TO VALIDITY

7.1 Construct Validity

There are questions concerning whether we are truly measuring just and all of the implicated code.

Bugs were collected from the Bugzilla databases for each project, and thus may not represent the complete set of all bugs. However, in all cases, Bugzilla is the designated repository of bug reports; we also focus on the fixed bugs, and thus the ones the community deemed worthy of repair.

We used an automated bug linking process which may be inaccurate. There may be both false positives and false negatives in the linked set. In a prior study [10] we evaluated the false positive and false negative rates and found them to be quite low. Bias in fix reporting [10] could have resulted in some fixes being unreport, and might have affected our results. In addition, our implicated code identification algorithm uses the `diff` tool. It is entirely possible that some of the changes in a revision marked as a bug fix are not, in fact, fixing lines which caused the bug. In lieu of this problem we use an approach used by well known prior studies [43]. Accuracy in identifying bug introducing changes may be increased by using advanced algorithms [29]. We use `git blame` which we have found produces highly accurate results, even in the presence of white-space changes and code movement [12]. We also ignore comments and blank lines in implicated code.

Labeling of bug severity may not be entirely accurate and depends on many factors including external factors such as bug reporter and internal factors such as the criticality of impact. However, we only focus on the fixed bugs, which the community deemed worthy of repair, and thereby we believe they are more accurate in their severity labeling. Availability of large number of bugs also provides better resilience against possible bias.

We compared properties of implicated code against randomly chosen background code. Our chosen random code distribution may not be representative of the implicated code distribution. We did take several measures, as described in Section 3.5 to mitigate this threat.

Sometimes "implicated code " may not really be buggy code, but rather a kind of "innocent bystander" that gets changed to remedy a flaw elsewhere, or even an unrelated change that just happened in a bug-fixing commit. However, one can reasonably expect that usually implicated code *is* indeed buggy code, and given the large number of fix inducing

hunks (ranging from 1,687 for Apache to 31,184 for Gimp) one can reasonably expect that the signals we see in the data are robust. Another issue is that bugs may have arisen due to *missing* code, such as an omitted `case` in a `switch`. We haven't addressed this issue.

For measuring general experience we consider only the contribution in that particular project. However, developers may contribute in several other projects, thereby garnering experience from those too. Measuring external experience would be difficult. However, given the long history of our studied projects and the participation of a large number of developers, it is reasonable to assume that our method reasonably estimates general experience of the developers.

7.2 Internal Validity

One issue (raised by one of the reviewers) is that implicated code is more likely to be "logically cohesive", and thus edited by a single author. This requires further study, beginning with a clear definition of "logically cohesive" code. Meanwhile, as seen with RQ1c above, even multiply authored implicated code generally shows higher ownership than background code.

We present evidence that implicated code tends to be associated with fewer authors. We also find that authors with prior experience of a file tend to be associated with less implicated code in that file. While strong correlation exists, the stringent requirements for causality have not been shown [27]. Despite this, our results do indeed show strong evidence that buggy code is different than other background code, and provide support for further research examining *why* it is different and whether these properties can be utilized to develop better process model or to predict software failures.

7.3 External Validity

In an attempt to address the generalizability of our findings, we have studied four real software projects that represent varying software processes and governance styles [9]. However, while it is reasonable to believe that our results are representative of open source software, it is unclear how well they generalize to commercial software, which may have different ownership policies and behavior. Again, we have provided evidence that bugs are different than background code, they have fewer number of authors and they are less likely to be introduced by the primary contributor of the target file.

All of our study subjects were written in C. While C is a very popular language, it is remarkably different from Object Oriented languages like Java. Given the rich encapsulation support of OOP, it is entirely possible that topic knowledge [38] would play a larger role and episodic knowledge [38] would be more reusable. It would be an interesting future work to study such possibilities.

8. CONCLUSION

We have studied 4 open source projects to understand impact of ownership and developer experience on software defects. Using version control history, we examine contributions to code fragments that are actually repaired to fix bugs. Are these "troubled" code fragments the result of contributions from many? or from one? Does experience matter? What type of experience? We find that implicated code is more strongly associated with a single developer's contribution; our findings also indicate that author's specialized experience in the target file is more important than general experience. Our findings suggest that quality control efforts could be profitably targeted at changes made by single developers with limited prior experience on that file.

Acknowledgments

We thank A. Bachmann and A. Bernstein for the bug linking data. We were supported by an IBM Faculty Fellowship, a gift from Microsoft Research and NSF grants SoD-TEAM 0613949 and SHF-Medium 0964703. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation. We gratefully acknowledge helpful feedback from anonymous ICSE reviewers.

9. REFERENCES

[1] Software errors cost u.s. economy $59.5 billion annually, http://www.nist.gov/public_affairs/releases/n02-10.htm.

[2] *Identifying reasons for software changes using historic databases*, volume 0, Los Alamitos, CA, USA, August 2002. IEEE Computer Society.

[3] *Succession: Measuring transfer of code and developer productivity*, Washington, DC, USA, June 2009. IEEE Computer Society.

[4] L. Argote. *Organizational Learning: Creating, Retaining, and Transferring Knowledge*. Kluwer Academic Publishers, Norwell, MA, USA, 1999.

[5] L. Argote, S. L. Beckman, and D. Epple. The persistence and transfer of learning in industrial settings. *Manage. Sci.*, 36(2):140–154, 1990.

[6] A. Bachmann and A. Bernstein. Software process data quality and characteristics: a historical view on open and closed source projects. In *IWPSE-Evol '09*, 2009.

[7] R. D. Banker, G. B. Davis, and S. A. Slaughter. Software development practices, software complexity, and software maintenance performance: A field study. *MANAGEMENT SCIENCE*, 44(4):433–450, April 1998.

[8] K. Beck. Embracing change with extreme programming. *IEEE computer*, 32(10):70–77, 1999.

[9] J. Berkus. The 5 types of open source projects, 2007. March 20, 2007 http://www.powerpostgresql.com/5_types.

[10] C. Bird, A. Bachmann, E. Aune, J. Duffy, A. Bernstein, V. Filkov, and P. Devanbu. Fair and balanced?: bias in bug-fix datasets. In *ESEC/FSE '09*, pages 121–130, New York, NY, USA, 2009. ACM.

[11] C. Bird, N. Nagappan, B. Murphy, H. Gall, and P. Devanbu. An analysis of the effect of code ownership on software quality across windows, eclipse, and firefox. Technical report, University of California, Davis, 2010. Submitted to FSE 2010. http://wwwcsif.cs.ucdavis.edu/ bird/.

[12] C. Bird, P. Rigby, E. Barr, D. Hamilton, D. German, and P. Devanbu. The Promises and Perils of Mining Git. In *MSR 2009*, 2009.

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

[14] F. Brooks. *The mythical man-month*. Addison-Wesley, 1995.

[15] 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.

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

[17] 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.

[18] B. Curtis, E. M. Soloway, R. E. Brooks, J. B. Black, K. Ehrlich, and H. R. Ramsey. Software psychology: the need for an interdisciplinary program. pages 150–164, 1987.

[19] 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.

[20] J. A. Espinosa. *Shared mental models and coordination in large-scale, distributed software development.* PhD thesis, Pittsburgh, PA, USA, 2002.

[21] M. Fischer, M. Pinzger, and H. Gall. Populating a release history database from version control and bug tracking systems. In *ICSM '03*, pages 23+, Washington, DC, USA, 2003. IEEE Computer Society.

[22] D. Freedman. Ecological inference and the ecological fallacy. *International encyclopedia of the social and behavioral sciences*, pages 4027–4030, 2004.

[23] T. Fritz, G. C. Murphy, and E. Hill. Does a programmer's activity indicate knowledge of code? In *ESEC-FSE '07*, pages 341–350, New York, NY, USA, 2007. ACM.

[24] T. Girba, A. Kuhn, M. Seeberger, and S. Ducasse. How developers drive software evolution. In *Eighth International Workshop on Principles of Software Evolution (IWPSE'05)*, pages 113–122, Washington, DC, USA, 2005. IEEE.

[25] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy. Predicting fault incidence using software change history. *IEEE Trans. Softw. Eng.*, 26(7):653–661, 2000.

[26] J. D. Herbsleb and A. Mockus. An empirical study of speed and communication in globally distributed software development. *IEEE Trans. Softw. Eng.*, 29(6):481–494, 2003.

[27] S. Kan. *Metrics and models in software quality engineering.* Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 2002.

[28] S. Kim, K. Pan, and E. E. J. Whitehead. Memories of bug fixes. In *SIGSOFT '06/FSE-14*, pages 35–45, New York, NY, USA, 2006. ACM.

[29] S. Kim, T. Zimmermann, K. Pan, and J. Jr. Automatic identification of bug-introducing changes. In *ASE '06*, pages 81–90, Washington, DC, USA, 2006. IEEE Computer Society.

[30] A. Meneely and L. Williams. Secure open source collaboration: an empirical study of linus' law. In *CCS '09*, pages 453–462, New York, NY, USA, 2009. ACM.

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

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

[33] N. Nagappan, B. Murphy, and V. Basili. The influence of organizational structure on software quality: an empirical case study. In *ICSE '08*, pages 521–530. ACM, 2008.

[34] M. Pinzger, N. Nagappan, and B. Murphy. Can developer-module networks predict failures? In *SIGSOFT '08/FSE-16*, pages 2–12, New York, NY, USA, 2008. ACM.

[35] G. P. Pisano, R. M. J. Bohmer, and A. C. Edmondson. Organizational differences in rates of learning: Evidence from the adoption of minimally invasive cardiac surgery. *Manage. Sci.*, 47(6):752–768, 2001.

[36] J. Ratzinger, M. Pinzger, and H. Gall. Eq-mine: Predicting short-term defects for software evolution. In M. B. Dwyer and A. Lopes, editors, *Fundamental Approaches to Software Engineering*, volume 4422 of *Lecture Notes in Computer Science*, chapter 3, pages 12–26. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[37] E. Raymond. *The cathedral and the bazaar: musings on Linux and open source by an accidental revolutionary.* O'Reilly & Associates, Inc. Sebastopol, CA, USA, 2001.

[38] P. N. Robillard. The role of knowledge in software development. *Communications of the ACM*, 42(1):87–92, January 1999.

[39] 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.

[40] M. A. Schilling, P. Vidal, R. E. Ployhart, and A. Marangoni. Learning by doing something else: Variation, relatedness, and the learning curve. *MANAGEMENT SCIENCE*, 49(1):39–56, January 2003.

[41] T. I. Seifert and B. Paech. Exploring the relationship of history characteristics and defect count: an empirical study. In *DEFECTS '08*, pages 11–15, New York, NY, USA, 2008. ACM.

[42] R. Shatnawi and W. Li. The effectiveness of software metrics in identifying error-prone classes in post-release software evolution process. *J. Syst. Softw.*, 81(11):1868–1882, 2008.

[43] J. Śliwerski, T. Zimmermann, and A. Zeller. When do changes induce fixes? In *MSR '05*, pages 1–5. ACM, 2005.

[44] D. Čubranić and G. C. Murphy. Hipikat: recommending pertinent software development artifacts. In *ICSE '03*, pages 408–418, Washington, DC, USA, 2003. IEEE Computer Society.

[45] D. G. Wastell. Learning dysfunctions in information systems development: overcoming the social defenses with transitional objects. *MIS Q.*, 23(4):581–600, 1999.

[46] E. Weyuker, T. Ostrand, and R. Bell. Do too many cooks spoil the broth? using the number of developers to enhance defect prediction models. *Empirical Software Engineering*, 13(5):539–559, October 2008.

[47] A. Zeller. *Why Programs Fail, Second Edition: A Guide to Systematic Debugging.* Morgan Kaufmann, 2 edition, June 2009.

[48] Y. Zhou and H. Leung. Empirical analysis of object-oriented design metrics for predicting high and low severity faults. *IEEE Trans. Softw. Eng.*, 32(10):771–789, 2006.