"Not My Bug!" and Other Reasons for Software Bug Report Reassignments

Philip J. Guo[1] pg@cs.stanford.edu
Thomas Zimmermann[2] tzimmer@microsoft.com
Nachiappan Nagappan[2] nachin@microsoft.com
Brendan Murphy[3] bmurphy@microsoft.com
[1] Stanford University, USA [2] Microsoft Research, USA [3] Microsoft Research, UK

ABSTRACT
Bug reporting/fixing is an important social part of the software development process. The bug-fixing process inherently has strong inter-personal dynamics at play, especially in how to find the optimal person to handle a bug report. Bug report reassignments, which are a common part of the bug-fixing process, have rarely been studied.

In this paper, we present a large-scale quantitative and qualitative analysis of the bug reassignment process in the Microsoft Windows Vista operating system project. We quantify social interactions in terms of both useful and harmful reassignments. For instance, we found that reassignments are useful to determine the best person to fix a bug, contrary to the popular opinion that reassignments are always harmful. We categorized five primary reasons for reassignments: finding the root cause, determining ownership, poor bug report quality, hard to determine proper fix, and workload balancing. We then use these findings to make recommendations for the design of more socially-aware bug tracking systems that can overcome some of the inefficiencies we observed in our study.

Author Keywords
Bug tracking, Bug triaging, Bug reassignment

ACM Classification Keywords
D.2.5 [Software Engineering]: Testing and Debugging; D.2.7 [Software Engineering]: Distribution, Maintenance, and Enhancement

ACM General Terms
Human Factors, Management, Measurement

INTRODUCTION
Bug reporting/fixing is a central part of the software development process, and one that always involves the coordination of multiple individuals. In large software projects (e.g., commercial ones like the Windows operating system or open-source ones like Eclipse or Firefox), the bug tracking system is the central hub for coordination, and the collection of informal notes about bug reports and development issues recorded within it form the main source of organizational memory [1] about the project's history. Developers often use bug tracking systems to perform expertise finding [2], making queries to determine who is the local expert on a certain software module or sub-system so that relevant questions and bug reports can be routed to him/her. For these reasons, we consider bug trackers to be one of the primary CSCW systems in software development.

The CSCW community has been interested in the social dynamics of bug fixing [3,4] and tools that improve the collaborative aspects of bug fixing [5]. But in general, most work on bug fixing have focused on how particular bugs should be fixed (or who is the best person to fix it) [6] and which types of bugs get fixed. To the best of our knowledge, there has been little work done on the social dynamics regarding *software bug reassignments*, a ubiquitous cooperative work activity mediated by a CSCW software system (the bug tracker).

For example, when a bug is assigned to someone, he/she can *reassign* it to someone else for reasons ranging from simply lacking the time to investigate deeply to a genuine attempt to find a person with better expertise. Figure 1 shows the number of reassignments versus the time until the bug report is first closed, for bugs in the Microsoft Windows Vista project. As the number of reassignments increases, we observe that the time required fixing a bug also increases. But contrary to popular belief, reassignments aren't necessarily 'bad', since it does take a few reassignments to find the true cause of a bug and who to properly fix it (it does take time, though). On the other hand, if there were few reassignments but the optimal bug fixer were not identified, then that could lead to a low-quality or faulty fix.

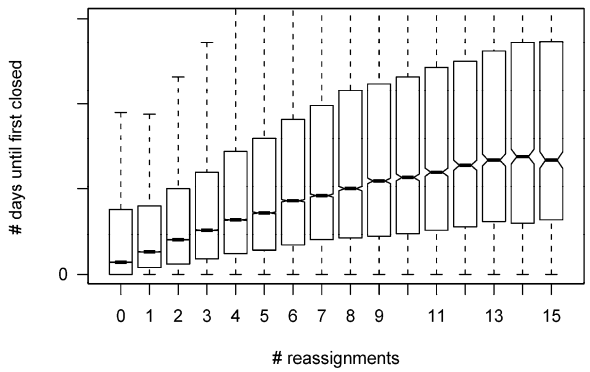

Figure 1. Number of reassignments vs. days until a Windows Vista bug report is first closed (y-axis hidden for confidentiality reasons)

CSCW 2011, March 19–23, 2011, Hangzhou, China.

Contributions: This paper presents a mixed qualitative and quantitative study of the collaborative aspects of bug report reassignments. Based on a widely-deployed qualitative survey at Microsoft, we categorized five primary reasons for reassignments: finding the root cause, determining ownership, poor bug report quality, hard to determine proper fix, and workload balancing. We then built and interpreted a descriptive statistical model to identify the relationship between bug report features and reassignments. We also show that there are certain harmful patterns of reassignments, like cycles at the end of a sequence. Finally, we use these findings to make recommendations for the design of more socially-aware bug tracking systems that can overcome some of the inefficiencies we observed in our study.

RELATED WORK

There has been a lot of work on bug triaging, not only in software engineering but also in the CSCW [4,3,5], HCI [7,8], and GROUP [9] communities. However to the best of our knowledge, there has been little work on bug report reassignments.

Reassignments in bug reports. Based on interviews with ten software developers and a qualitative analysis of an unspecified number of bug reports, Halverson et al. [5] described several problematic patterns in bug tracking. They observed that assign/reassign cycles (or "ping pong" as called by developers) indicate that a bug is not finding the right owner or that the location of the bug is ambiguous. In an empirical study of which bugs get fixed in Microsoft Windows, Guo et al. [10] observed "reassignments are not always detrimental to bug-fix likelihood; several might be needed to find the optimal bug fixer." Compared to this previous work by Halverson et al. and Guo et al., this paper provides a *comprehensive discussion* of causes for reassignments. Our findings are *quantitatively validated on a large number of bug reports*.

Jeong et al. [11] analyzed bug report reassignments (which they called "bug tossing") in the Mozilla and Eclipse projects. They used a graph structure and Markov chains to reduce the number of reassignments. Jeong's work was inspired by Shao et al. [12], who proposed an algorithm for ticket routing. In ticket routing, a new ticket needs to find its resolver with as few steps as possible—any assignment to someone who cannot resolve the ticket is considered as inefficient. However, as we show in this paper, there are many legitimate reasons for bug reassignments, such as *finding the root cause* and *workload balancing*.

Communication, coordination, communities, and bugs. Several researchers investigated how people communicate and coordinate in bug reports. In his Ph.D. thesis, Sandusky used qualitative methods on open-source bug reports for an empirically grounded description of the information practices used by a distributed open-source project [13]. Sandusky and Gasser studied the role of negotiation and its effect on the organization of information in software problem management [9]. Ripoche and Sansonnet analyzed speech acts across the Mozilla corpus of bug reports [14]. Breu et al. [3] categorized questions asked in open-source bug reports and analyzed response rates and times. Carstensen studied coordination via physical bug forms [15].

Aranda and Venolia [16] reported on a study of coordination activities around bug fixing at Microsoft. They identified common coordination patterns and provided implications for tool designers and researchers. Bertram et al. [4] conducted a qualitative study of issue tracking systems as used by small, collocated software development teams. They found that even in collocated teams, issue trackers are a focal point for communication and coordination. Ko and Chilana [7] quantified the value of contributions by "power users" to open bug reporting in Mozilla. They observed that the primary value comes from recruiting a small pool of talented developers and reporters, and not from the masses. Diederik van Liere [17] studied how the information provided by open-source community members influences the repair time of software defects; he found that user contributions shorten repair times.

Characterization of bug reports. Ko et al. [18] looked at bug report titles and identified fields that could be incorporated into new bug report forms. Bettenburg et al. [19] conducted a survey among developers and users from the Apache, Eclipse, and Mozilla projects to determine which information contents comprise good quality bug reports. Just et al. [20] analyzed the responses from the same survey to suggest improvements to bug tracking systems.

METHODOLOGY

We studied bug report reassignments in the context of the Microsoft Windows Vista operating system project, which we feel is a representative example of a large-scale commercial software project. Vista contains several thousand source code files and 40+ million lines of code, written by more than 2000 software engineers. The findings we present in this paper are derived from three sources related to Windows Vista bug reports: free-response answers from a survey sent to Microsoft employees, a manual examination of randomly-selected bug reports, and a high-level quantitative analysis of the entire Windows Vista bug database.

Survey free-response answers

Our primary data source is an online survey we sent in August 2009 to 1,773 Microsoft employees with questions about various aspects of the bug triaging and fixing process. Since we wanted to get the opinions of people well-versed in handling Windows-related bugs, we chose as our survey participants the top 10% of people who have opened, been assigned to, or resolved Windows Vista bugs. We received 358 responses (20% response rate). Most respondents were either developers (55%) or testers (30%). Most were fairly experienced, with a median of 11.5 years of work experience in the software industry and 9 years at Microsoft.

We analyzed responses to most of the survey questions for another paper [10]; for this paper, we analyzed responses to the following free-response question, which we did *not* explore in our other paper:

> In your experience, what are some reasons why a bug would be **reassigned multiple times** before being successfully resolved as *Fixed*? E.g., why wasn't it assigned directly to the person who ended up fixing it?

Response length varied from one phrase (e.g., "bug cause was not initially understood") to long paragraphs. We printed out all 358 responses on index cards and performed card sorting [21]. Two of the authors independently performed an open card sort and then merged their results into a single taxonomy. Then a third author read over all of the responses to check and made minor adjustments to the categories.

Manual examination of bug reports

Informed by our analysis of survey results, we informally examined the contents of 50 Windows Vista bug reports, chosen by randomly sampling from all bug reports with more than 5 reassignments (10% of total bug reports had more than 5 reassignments). The main reason we manually examined selected bug reports was to corroborate the survey respondents' opinions with firsthand observations from the bug reports themselves.

Quantitative analysis of bug and personnel data

We quantified certain observations to the extent possible by mining data from the Windows Vista bug database and the Microsoft employee personnel database. We collected all pre- and post-release bug reports for Windows Vista in July 2009 (2.5 years after Vista's release date). We consider our dataset to be fairly complete for the factors we want to investigate, since very few new Vista bugs are being opened, compared to when it was under active development (2002-2007). For confidentiality reasons, we cannot reveal the exact number of bug reports, but it is at least an order of magnitude larger than datasets used in related work [22]. For each bug report, we extracted a list of edit events that occurred throughout its lifetime. Each event alters one or more of the following fields (fields not relevant to our analysis in this paper have been omitted):

- **State:** OPENED, RESOLVED, or CLOSED
- **Opener:** Who opened this bug?
- **Assignee**: Who is now assigned to handle this bug?
- **Severity:** An indicator of the bug's potential impact on customers. Crashes, hangs, and security exploits have the highest severity (Level 4); minor UI blemishes, typos, or trivial cosmetic bugs have the lowest severity (Level 1).
- **Component path:** Which component is the bug in? e.g., `DesktopShell/Navigation/StartMenu`
- **Bug type:** What kind of bug is it? e.g., bug in code, specification, documentation, or test suite
- **Bug source:** How was this bug found? e.g., by a customer, an internal Microsoft user, or a system test
- **Resolution status:** How has this bug been resolved? e.g., FIXED, BY DESIGN, WON'T FIX, NOT REPRODUCIBLE. (Null if state is not RESOLVED)

Here is a typical bug's life cycle: When it is first opened, all of its fields except for "Resolution status" are set. Then the bug might be edited a few times (e.g., to upgrade its severity). A special type of edit called a **reassignment** occurs when the "Assignee" field is edited. When somebody thinks that he/she has resolved the bug, its "Resolution status" field is set. After the resolution attempt is approved (usually by the opener), the bug is closed. However, it might be reopened if the problem has not actually been properly resolved.

To explore the impacts of geographical and organizational distance on bug reassignments, we obtained the office location and manager of each employee circa July 2009 from the Microsoft employee personnel database. Thus, we can determine whether two employees worked in the same building, campus, country, or on the same team (i.e., had the same manager). Sometimes people switch locations or teams, but in general Microsoft tries to keep employees in the same location and team during a product cycle [23].

Follow-up survey

Lastly, we solicited additional feedback on our findings in another survey among 397 Microsoft employees. We chose as our participants based on the number of bug reports they have been assigned to or the number of reassignment cycles they have been involved in Windows 7. We received 118 responses (30% response rate)

CAUSES OF BUG REPORT REASSIGNMENTS

In this section, we combine card sort results from our survey, observations from examining selected bug reports, and descriptive statistics to characterize the 5 main causes of bug reassignments in the Windows Vista project:

1. Finding the root cause
2. Determining ownership
3. Poor bug report quality
4. Hard to determine proper fix
5. Workload balancing

A typical bug report gets reassigned a few times before it gets resolved. The median number of reassignments for Windows Vista bug reports is 2, and the mean is 2.5. 90% of reports have 5 or fewer reassignments. In general, reassignments aren't necessarily detrimental, but they do take up time and cause developers to context-switch between multiple reports.

Finding the root cause

The most common reason bug reports are reassigned is because people want to find the root cause of the problem before they are willing to attempt a fix. Bug reports usually only indicate superficial symptoms, but a high-quality fix should address the root cause and not merely patch the reported symptoms. The root cause is often in a completely different component than symptoms indicate, though. A

survey respondent elaborates on this reason for why bugs are reassigned multiple times before being resolved:

> *"Bugs many times are exposed in the UI [user interface], but are not caused by the team writing the UI code. These bugs can pass down several layers of components before landing on a lower level component owner. As the UI team gets more familiar with the component layers they can more directly assign bugs to the offending component, but that takes time and knowledge."*

We can quantify the above phenomenon by correlating reassignments with changes in the "Component path" field of bug reports, which indicates in which component people currently believe a bug originates. People don't usually change a bug's component path without also reassigning it: If a bug report had no reassignments, then it only has a 13% chance of its component path being changed, while a bug with some reassignments has a 35% chance of its path being changed (almost 3x more). There is a Spearman's rank correlation [24] of 0.32 between the numbers of reassignments and path changes for individual bug reports, which indicates a moderate positive correlation.

Oftentimes the bug reporter doesn't have the expertise required to ascertain the root cause, so he/she must reassign the bug to someone with more domain-expertise. As a survey respondent describes:

> *"Usually this seems to stem from inaccurate assumptions on the part of the bug filer. For example, someone clicks a button in a feature, and there's a corresponding crash — usually the bug is assigned to the most proximal piece of interaction — the button owner. However, given software complexities, sometimes the crash is actually due to an underlying layer. The filer either lacked the expertise, will, or time to investigate deep enough to understand the issue at hand."*

Our data corroborates these anecdotal observations: Bugs originating from different sources have different average numbers of reassignments. On one end, internal users (Microsoft employees using beta versions) have a hard time reporting bugs to the right components, thus resulting in the most reassigns of any bug source (mean of 3.14, median of 2). For example, one of the authors of this paper (a Microsoft employee) once reported a bug for Microsoft Office, but since he did not work on the Office project, it was hard for him to determine which exact component to file the bug under. On the other extreme, bugs found by component and system tests have relatively few reassigns (mean of 2.4, median of 1), since they are purposefully designed to isolate particular components, so their root causes are quite certain.

Unfortunately, reassignments are also done out of laziness; some people don't do a thorough job of determining root cause and simply punt the bug to get it off their task queue:

> *"Insufficient root cause analysis. People are willing to do just enough to convince themselves it isn't their problem and then re-assign to the person who they think is closer to the right owner."*

At the end of this paper, we make design recommendations for improving expertise finding [2] and thereby minimizing the number of reassignments required to ascertain a bug's root cause.

Determining ownership (which is often unclear)

A concept related to root cause is 'ownership', which is defined roughly as "what team is responsible for the component that exhibits this bug?" In a large software project like Windows Vista, ownership of components can often be unclear or ambiguous, since many components lie at the intersection of several teams' jurisdictions. These survey respondents lament:

> *"It is often very difficult to identify the correct owner for the bug, even when the cause of the bug is known."*

> *"The bug falls into an area between two teams. Say, the USB team and the WPD (Windows Portable Devices) team. The bug gets kicked around many times while the teams decide who is actually at fault."*

When we manually looked through bug reports, we saw these disagreements over ownership play out in their edit histories. As an example of such a scenario, in one bug report, Person A first assigns to Person B, with the message "*You or [Person C]?"* An hour later, Person B reassigns to Person C with the message *"Reassign to [Person C] …"* along with a brief explanation of why he thought that the bug was in a component that Person C owned. The next morning, Person C reassigns *back to* Person B with the message *"Dunno who gets this one, but it's not me. I don't have anything to do with [Component X], AFAIK* [as far as I know]*."* After another day of investigation, Person B then reassigns to Person D, who works on the bug for 2 weeks and then successfully resolves it as "Fixed".

When there are disagreements over ownership, bugs can be reassigned back-and-forth between two (or more) teams, an undesirable, time-wasting phenomenon known to our survey respondents as "bug pong" or "hot potato":

> *"Not clear ownership: Sometimes different teams work together to develop a product. In such cases sometimes the ownership boundaries are not clear so the bugs get re-assigned back and forth till the ownership gets determined."*

> *"Playing bug pong between teams who don't agree on ownership. It's stupid, but some teams use this as a delaying-until-it's-bad-enough-that-someone-more-important-demands-a-fix."*

In the follow-up survey, we also asked about the frequency of hot potatoes. The majority of respondents replied that hot potato is "uncommon". Yet some respondents pointed out situations where hot potatoes occur frequently: for components shared by multiple teams, high in the system stack, or

with unclear ownership; near milestones; or for bugs with incomplete steps to reproduce.

At the end of this paper, we make recommendations for making ownership more explicit so as to reduce the amount of these inefficient reassignments.

Poor bug report quality

If a bug report is poorly written or contains too little information, then it might need to be reassigned a few times as people struggle to decipher its cause:

> *"The most important factor in multiple reassigning in my experience is unclear bug reports. If the person assigned to the bug doesn't understand the issue, they will either assign it back to the person who opened it, or (rarely, but it happens) assign it to the wrong person based on misunderstood information, and then it will become even worse."*

> *"If a bug report cites only basic symptoms (such as 'crash') and has little or no information hinting at cause (such as call stack), then triage is very difficult and a bug can end up being bounced around."*

When we manually looked through bug reports, we saw the detrimental effects of poor report quality in some of their edit histories. An example scenario: For a particular user interface bug, Person A first assigns to B with the 2-word bug report *"please investigate"* without providing much further detail. Person B investigates for ~2 hours and reassigns to Person C with the message "*I debug to [function F]. I cannot match the source code. Before this function return, No permission dialog pops up. Please take a look."* Person C *immediately* reassigns back to B with the message *"um ... [function F] is the guy who's rendering the dialog. If you're complaining about the dialog you should find out who requested the dialog to show up."*

It's difficult to quantify bug report quality without doing some sort of heuristic-based text analysis that is outside the scope of this project; however, one proxy indicator of poor report quality is that a bug report's "Bug type" field changes throughout its lifetime. If people aren't even sure about the type of the bug (e.g., is it a bug in code, specs, docs, or tests?), then chances are that it's a poor-quality bug report. 9% of all Windows Vista bugs had their bug type field changed. Bugs whose type changed had, on average, more reassigns than those whose types didn't change: mean of 3.6 reassigns vs. 2.4, and median of 3 vs. 2.

Hard to determine proper fix even after cause known

Even after the root cause and ownership have been determined, a bug might *still* need to be reassigned as people debate the proper way to fix it. As our survey respondents observed:

> *"There can be multiple possible fixes for a given issue which can straddle teams, so the bug can bounce back and forth until the bug fix strategy is solidified."*

> *"Bug could be fixed or worked around in multiple places, and each place punts the fix to one of the other teams."*

Workload balancing (or the appearance thereof)

Once a bug report gets to the proper team that is eventually going to fix it, it *still* might get reassigned a few times between team members as a matter of workload balancing. For example, some developers might be busy with other tasks, so they will reassign to their teammates (with the hopes of reciprocity in the future). Such load-balancing reassignments can be beneficial, since bugs might get fixed sooner:

> *"Once the bug has found the right team, the biggest factor in reassigning is often load balancing issues across team members to drive down totals. Bugs will be fairly static early in the development cycle but as bug counts become more important, we'll move issues around frequently to ensure they get prompt attention."*

However, sometimes within-team reassignments are done for political reasons, giving the appearance of being load-balanced to satisfy managers while the bug sits idle ('parked'):

> *"A bug is parked with someone. This may be for investigation. It may be for some desire to appear load balanced. I believe reassignment is more common when playing games with balancing than it is when investigation finds that the responsible code is owned by another individual to whom the bug is transferred."*

At the end of this paper, we make recommendations for monitoring developer activities in order to facilitate load balancing.

DESCRIPTIVE STATISTICAL MODEL

To quantify factors that contribute to bug reassignments, we built a descriptive statistical model and interpreted its coefficients with reference to the qualitative findings we presented in the previous section.

Logistic regression model building

Specifically, we built a logistic regression model for the probability that a bug report has excessive numbers of reassignments, where we define "excessive" as greater than 5. We used 5 as our cutoff threshold since 90% of bugs had 5 or fewer reassignments, so those with greater than 5 can be thought of as having "excessive numbers of reassignments" (in the top 10%). We used a cutoff since we only wanted to separate reports with "normal" and "excessive" numbers of reassignments; it didn't make much sense to try to predict the *exact number* of reassignments (e.g., it doesn't matter if a bug has 23 or 42 reassignments; both are "excessive").

A logistic regression model aims to predict the probability of an event occurring (e.g., does this bug report have excessive numbers of reassignments?) using a combination of factors that can be numerical (e.g., number of component path changes), Boolean (e.g., was its severity level upgraded?), or categorical (e.g., bug source).

Table 1 shows the model we constructed by training on the entire Windows Vista bug report dataset using the R statistics package. We determined that all factors had independent effects by adding each one to an empty model and observing that the model's deviance (error) decreases by a statistically significant amount for all added factors (a standard technique called *Analysis of Deviance* [25]).

Note that the sole purpose of our model is to *describe* various independent effects on bug reassignments. It cannot actually be used in practice to *predict* the probability that a newly-opened bug report will have excessive (greater than 5) reassignments, since it uses factors that are not available at the time a bug is first opened (e.g., number of component path changes).

How to interpret logistic regression coefficients

One main benefit of using logistic regression over other types of statistical models (e.g., support vector machines) is that its parameters (e.g., the coefficients in Table 1) have intuitive meanings.

For **numerical and Boolean factors**, the sign of each coefficient is its *direction* of correlation with the probability that a bug contains excessive reassignments. For example, "Num. component path changes" is positively correlated with reassignments, so its coefficient is positive (0.72). The magnitude of each coefficient approximately indicates how much a particular factor affects reassignments. See Hosmer and Lemeshow [25] for details on how to transform these coefficients into exact probabilities. In general, it's hard to compare coefficient magnitudes across factors, since their units of measurement likely differ. However, it's possible to compare coefficients for, say, two Boolean factors like "Severity level upgraded?" and "Bug type changed?" The coefficient of the former (1.30) is larger than that of the latter (0.87), which means that a severity upgrade has a *larger positive effect* on the probability that a bug will have excessive reassignments than a change in bug type does.

For **categorical factors** ("Bug source" is the only one in our model), if a factor has N categories (levels), then $N - 1$ of them get their own coefficient, and the remaining one gets its coefficient folded into the intercept term (the R statistics package we use chooses the alphabetically earliest category to fold, so that's why "Ad-hoc testing" has no coefficient in Table 1). What matters isn't the value of each coefficient but rather their *ordering* across categories. For example, "Internal user" has a larger coefficient than "Human review", which means that the former is more positively correlated with reassignments than the latter.

Interpreting our model's coefficients

Bug source is a categorical factor whose coefficients can only sensibly be compared against one another. Bugs reported by internal Microsoft users are likely to have excessive reassignments, since Microsoft employees using beta versions of software have permission to directly submit bug reports but often lack the expertise to submit a high-quality report targeting the specific component exhibiting the bug.

Factor		Coefficient
Bug source: (categorical)	Internal user	0.26
	Component test	0.11
	System test	0.11
	Human review	0.05
	Ad-hoc testing	†
	Code analysis tool	*
	Customer	*
Num. component path changes		0.72
Initial severity level		0.15
Severity level upgraded? (Boolean)		1.30
Bug type changed? (Boolean)		0.87
Bug opener reputation		-0.16
Opener / 1st assignee same manager		-0.52
Opener / 1st assignee same building		-0.26

Table 1. Descriptive logistic regression model for whether a bug report has greater than 5 reassignments, trained on all Windows Vista bugs. Factors labeled * had statistically insignificant coefficients (with p > 0.001), so they cannot be meaningfully compared. The factor labeled † folds into the intercept term, which is omitted for confidentiality.

In contrast, QA staff usually vet bugs submitted by customers before entering them into the bug database. Bugs found by component and system tests are less likely to be reassigned since it's much easier to pinpoint their root causes and ownership; after all, tests are designed to target specific well-defined areas. Finally, bugs found by human review (e.g., code or documentation review) are unlikely to have excessive reassignments, since if a bug is found in someone's code or documentation during a review, then *they* are likely the owner responsible for fixing it.

Number of component path changes is positively correlated with excessive reassignments, since a bug's component path changes throughout the normal process of ascertaining root cause and determining ownership.

Initial severity level is positively correlated with excessive reassignments, since higher-severity bugs get more attention so people might pass them around in an effort to triage and fix them. In contrast, many low-severity bugs are simply 'parked' in someone's task queue and receive little attention (since they are probably busy handling higher-severity bugs).

If a bug's **severity level is upgraded**, then that's a strong "call to action" for developers to work harder to find the root cause, assign ownership, and actually fix the bug. Thus, it's also positively correlated with reassignments.

If a bug's **type is changed**, then it's likely a low-quality bug report (it doesn't even contain enough information for people to accurately determine its type), which our survey respondents mentioned was positively correlated with reassignments.

The **bug opener's reputation** is negatively correlated with reassignments. We quantify reputation using the same metric as Hooimeijer and Weimer [22]:

$$\text{bug opener reputation} = \frac{|\text{OPENED} \cap \text{FIXED}|}{|\text{OPENED}| + 1}$$

For each bug report, we calculate its opener's reputation by dividing the number of *previous bugs* that he/she has opened and gotten successfully fixed by the total number of previous bugs he/she has opened (+1). Adding 1 to the denominator prevents divide-by-zero and, more importantly, prevents people who have opened very few bugs from earning high reputations (e.g., 1/(1+1) << 100/(100+1)). Bug openers with higher reputations (i.e., those better at getting their bugs successfully fixed) might be more experienced in finding the right person to assign bugs to, thus not incurring as many reassignments.

If the bug's opener and first assignee have the **same manager** (i.e., are on the same team), then the bug is less likely to have excessive reassignments. Bugs assigned between team members get the benefits of better communication and more face-to-face discussions rather than having disagreements recorded in the bug database as reassignments.

Similarly, if the bug's opener and its first assignee work in the **same building**, then the bug is also less likely to have excessive reassignments, again due to the benefits of face-to-face contact.

QUANTIFYING REASSIGNMENT PATTERNS

We performed a quantitative analysis to explore the question of whether certain *patterns* of reassignments (e.g., cycles or back-and-forth "bug pong") had an impact on the chances that a bug gets *successfully* fixed. By "successfully fixed" we mean that its final resolution status is FIXED (as opposed to an unsuccessful resolution status like BY DESIGN, WON'T FIX, or NOT REPRODUCIBLE).

Certain patterns of reassignments are beneficial to bugs getting *successfully* fixed (so they are "good reassignments"), while others are detrimental ("bad reassignments"). Thus, in order to improve the chances that a bug will be successfully fixed, we should strive to make recommendations to encourage "good reassignments" while discouraging bad ones (not to merely reduce the total number of reassignments).

Reassignment cycles at the beginning of triage

We observed that reassignment cycles at the *beginning* of the triage process are beneficial for getting a bug successfully fixed. By 'cycle' we mean reassignment back to a person who has previously been assigned the bug, thus forming a cycle in the sequence of assignees.

For concreteness, let's use x to denote the base probability of any Windows Vista bug being successfully fixed (we cannot reveal the exact value of x due to confidentiality reasons). Let's use sequences of letters to denote reassignment patterns: e.g., "ABA" means the bug is first assigned to Person A, who then reassigns it to Person B, who then reassigns it back to Person A.

Cycle size	Beginning	Middle	End
2	1.11x	1.05x	0.96x
3	1.10x	1.06x	0.96x
4	1.12x	1.06x	0.93x
5	1.04x	1.03x	0.89x
6	1.07x	1.01x	0.97x
7	1.03x	0.99x	0.88x

Table 2. The effects of cycle size and location on the likelihood of a bug report being successfully fixed. The exact percentages are confidential, so we present values normalized relative to x, which is the likelihood of successful fix for any bug report with at least one cycle.

An ABA sequence at the beginning of triage has a 1.16x chance of getting successfully fixed: 16% greater than the baseline. In contrast, an ABC sequence has a 1.05x chance, and an AB[END] sequence (Person A assigns to Person B, and then the investigation stops) has only a 0.96x chance of being successfully fixed. Thus, it's better to have a cycle at the beginning of triage (ABA) than to pass it onto a new person (ABC) or simply ending the investigation.

This same pattern holds true for sequences of length 4: ABCA has a 1.11x chance of successful fix, ABCB has a 1.08x chance, ABCD has a 1.06x chance, and ABC[END] has only a 1.02x chance. Again, the presence of a cycle (ABCA and ABCB) is more beneficial than its absence (ABCD and ABC[END]).

In fact, the benefits of cycles at the beginning of triage are present even as the cycle size increases. Table 2 shows the relative chances of a bug being successfully fixed if it contains cycles of sizes 2 through 7. For reference, ABA is a cycle of size 2, while ABCA is a cycle of size 3. The leftmost "Beginning" column shows that cycles at the beginning of triage are better than those in the middle or at the end. Furthermore, all bugs containing cycles at the beginning have greater than the baseline x chance of being successfully fixed.

The respondents of the follow-up survey pointed out that the main reason for beneficial cycles in the beginning is that if the *initial* assignee passes the bug onto someone else but then it gets *back* to him/her, there is now more information to effectively fix the bug rather than give up on it.

> *"The initial bug report is incomplete or inaccurate and Alice sends back to the tester (Bob) for more information, better repro steps, etc. This is a common cycle. Once the bug is improved, it has a high likelihood of being fixed."*

Respondents also pointed out that cycles occur often when someone is searching for the correct owner of a bug report. Such a cycle in the beginning indicates that while Bob was not the actual owner, he probably provided some pointers to Alice on who can fix the bug.

Reassignment cycles at the end of triage

In contrast, Table 2 shows that reassignment cycles at the *end* of the triage process are detrimental to the chances of a bug being successfully fixed. An example of a cycle at the end of the triage process is ABCDEFGF[END], where FGF is a cycle of size 2 at the end of triaging. All entries in the rightmost "End" column have less than the baseline *x* chance of being successfully fixed.

The respondents of the follow-up survey pointed out that the main reason for detrimental cycles at the end (whereas cycles in the beginning are beneficial) is that they are related to discussions whether a bug should be fixed at all.

> *"This example feels more like a triage cycle where Alice is the PM [program manager] (or opener) and Bob is the war team/triage team, etc. The war team is sending the bug back to PM/opener for justification why the bug should be fixed (and not punted). The fact that this conversation is happening at all means the bug is at risk and likely to be punted."*

Unclear ownership was another reason mentioned occasionally in the responses to the follow-up survey:

> *"When ABA is at the end, I think the bug is likely going back and forth between two developers, who either do not agree, or do not want the responsibility of fixing the bug."*

THREATS TO VALIDITY

Internal validity: In a qualitative study of ten bugs, Aranda and Venolia [16] found that sometimes details are discussed even before a bug report is created and that not all information is recorded in bug tracking systems. For our study, this is only a minor threat because bug reassignments must be recorded in the bug database. We also validated our quantitative results with qualitative feedback from Microsoft employees.

Bird et al. [26] raised the issue of bias in bug datasets for defect prediction in open-source projects. However, the likelihood of bias in our dataset is low since we analyzed the entire population of Windows Vista bug reports.

External validity: Drawing general conclusions from empirical studies in software engineering is difficult because any process depends on a potentially large number of relevant context variables [27]. For this reason, we cannot assume a priori that the results of our study generalize beyond the specific environment in which it was conducted. That is, other large-scale systems software projects.

LESSONS LEARNED AND RECOMMENDATIONS FOR BUG TRACKING SYSTEM DESIGN

Not all reassignments are necessarily bad

Previous research [5,11] considered all bug reassignments to be problematic and consequently proposed ways to avoid reassignments. However, as the study in this paper shows, bug reassignments are often needed to locate the root cause and the person who should fix the bug. Unfortunately, it is not yet possible to automatically separate the wheat (beneficial reassignments) from the chaff (unnecessary reassignments). While in some cases, it is possible to identify problematic patterns such as "ping pong" bugs [5], such patterns typically apply to only a small fraction of bug reports. In the follow-up survey, most respondents considered ping pong bugs to be fairly uncommon. We also asked about the percentage of detrimental/wrong reassignments. On average respondents considered only 17.6% of reassignments to be detrimental; the median was even lower with 10%.

Ideally, ***bug tracking systems would have ways to assess and rate reassignments***. Beneficial reassignments could be marked by users or automatically identified with heuristics. This would help to increase the quality of tools that leverage reassignment information to make recommendations to engineers. Bug tossing graphs [11] are an example of such a tool, which can reduce the number of reassignments. However, bug tossing graphs do not have the concept of beneficial reassignment; their goal is simply to direct a bug report to the final resolver via as few intermediate people as possible. Thus, it is possible and likely that an essential person is omitted from the list of people who inspect a bug report.

Tool support for finding root causes and owners

A salient finding from our study is the significance of root causes and component owners when fixing bugs. Often it is not immediately clear from the bug description which part of the software needs to be fixed; bug reassignments narrow down possibilities for fault location. Once the fault location is known, another challenge is identifying the right person who is able to fix the fault; again this can lead to reassignments because ownership is not always clearly defined. Based on this observation, we make several recommendations for improving bug tracking systems:

1. Integrate a ***knowledge database of top experts*** and their areas of expertise into bug tracking software. For example, recommending the best engineers to fix heap corruption errors would allow other engineers to assign specialized types of bug reports to the people who are most skilled to either fix the bug report or to find someone who can.

2. Similarly, having experienced technical engineers on the team who are intimately familiar with the entire module's code base and can pick the right engineers to work on bugs, will help to reduce the number of misdirected assignments. While several projects have engineers responsible for bug triaging, especially in open-source projects [6,28], there is only limited tool support in existing bug tracking systems related to bug triaging.

   Ideally, ***bug triagers act as information hubs*** and are aware of the entire social network of engineers and the technical dependency network. To support engineers staying on top of these networks, tools and techniques from the field of socio-technical congruence [29,30] should be integrated into bug tracking systems.

3. Once the fault location has been narrowed down, ***better tools for finding code ownership and expertise*** based on actual code contributions would help in identifying the appropriate person who can resolve the bug report and avoiding unnecessary reassignments. Note that in practice, ownership and expertise are often two different concepts. Someone who owns a piece of code might not necessarily have the most expertise to change it. While it is difficult to mine ownership automatically, several approaches can identify engineers who are familiar with a piece of code [31,32,2].

Assign bugs to arbitrary artifacts rather than just people

Another more radical change to bug tracking is to allow assignment of bug reports to *one or more arbitrary artifacts* rather than just one person. Examples of artifacts include components, files, but also UI elements, features, or simple keywords. Based on historical data and social networking techniques or expertise finding techniques [32,2], keywords could then *fluidly* map to people who probably can fix the bug. For example, engineers who previously have fixed bug reports about keyword *WindowManager* will see any new bug reports about this keyword (and related keywords).

This extra layer of indirection means that bug reports can be assigned to multiple persons rather than individuals. While this might come at the cost of lower accountability, we believe that more bug reports will find the right person faster. Rather than developers fixing bugs reactively when assigned reports, the role of developers would be more proactive, constantly picking bug reports from a pool. If certain reports are not picked after a certain amount of time, they could be automatically assigned to the most appropriate developers, based on heuristics.

Tool support for awareness and coordination

Another recommendation is to ***increase the awareness of the changes happening around bug (re)assignments***. For example, if Person A assigns to B, but then B assigns to C, then A typically does not know that B assigned the bug to C, and would be under the impression that B should get future bugs (of that type or component) when in fact C should be assigned those bugs. If Person A were more aware of the updates to reassignments, that could help better direct his/her own future reassignments.

Bug tracking systems should also include ***better visualizations of reassignment patterns*** to help engineers identify problematic patterns such as reassignment cycles or ping pongs. Similar to context awareness, a visualization of the status of the bug reassignment would help engineers understand the process of finding the right engineer for a bug so that this knowledge can be applied to future bugs. Halverson et al. [5] proposed visualizations for the history of individual work items and the social health of all open work items in a project. Their primary focus was to identify problematic patterns. Ultimately, we believe that the way that engineers interact with bug reports needs to move away from a bug list and to-do list to more flexible presentation. One of these presentations might consist of (code) bubbles [33,34]. A bubble is a fully editable and interactive view of an artifact that exists in a large, pannable 2-D virtual space.

Furthermore, information on ***bug reassignments can be used by engineers for archival purposes*** too. For example, if an engineer wants to find out who should be assigned bugs that are part of component X, he/she can extract the bugs from the database and look though the reassignment patterns to gain a better understanding of the correct person to assign the bug to. Currently, the reassignment information in bug databases is simply presented as a series of text fields and edits, which is hard to decipher and makes it cumbersome to extract high-level patterns. We feel that historical reassignment data should be easily accessible for engineers to make the right triaging decisions.

Finally, most ***bug tracking systems measure only when a person edits a bug report, but not when they are in the process of investigating the report***. To increase workload awareness, we recommend building a system that would let developers/testers pick a bug they plan to work on and have the system to passively (unobtrusively) monitor their activity while they work on that bug. This way, team members and managers will know if a developer is actively working on a bug or whether the bug is parked (inactive). This will allow team members and managers to find out if a developer is already overloaded, so that they will know to find alternative options to fix this bug.

CONCLUSION

In this paper, we have investigated the bug reassignment process in Windows Vista using qualitative and quantitative approaches. To the best of our knowledge, our paper is the first to study these social dynamics in the bug reassignment process. In sum, we learned that:

- Reassignments are not always harmful. They are in fact beneficial to find the best person to fix a bug. Excessive reassignments are harmful, though.
- Qualitatively, the five primary reasons for reassignments are finding the root cause, determining ownership, poor bug report quality, hard to determine proper fix, and workload balancing.
- Quantitatively, the number of component path changes, initial severity level, upgrading the severity level, and bug type change correlate positively with reassignments, whereas the bug opener's reputation and co-location of opener and first assignee correlate negatively.
- Based on quantifying reassignment patterns, we observe that cycles at the beginning of bug triage are useful for finding the right person to fix the bug, but cycles at the end are detrimental.

Bug reassignments currently occur in an ad-hoc manner as part of the software development process. There is little tool support in current bug tracking systems for efficiently directing reassignments. We hope that designers of future bug tracking systems can adopt our recommendations to create more socially-aware systems that, amongst other goals, eliminate inefficient reassignments.

Acknowledgments: Thanks to the CSCW reviewers for their insightful critiques and to the Microsoft Windows team for their help in understanding the data. Philip Guo performed this work during a summer internship and a visit to Microsoft Research.

REFERENCES

1. Randall, D., Jon, O., Rouncefield, M., and Hughes, J.A. Organizational Memory and CSCW: Supporting the Mavis Phenomenon. In *Proceedings of the 6th Australian Conference on Computer-Human Interaction* (1996), 26-33.
2. McDonald, D.W. and Ackerman, M.S. Expertise recommender: a flexible recommendation system and architecture. In *Proceeding on the ACM Conference on Computer Supported Cooperative Work* (2000), 231-240.
3. Breu, S., Premraj, R., Sillito, J., and Zimmermann, T. Information needs in bug reports: improving cooperation between developers and users. In *Proceedings of the ACM Conference on Computer Supported Cooperative Work* (2010), 301-310.
4. Bertram, D., Voida, A., Greenberg, S., and Walker, R. Communication, collaboration, and bugs: the social nature of issue tracking in small, collocated teams. In *Proceedings of the ACM Conference on Computer Supported Cooperative Work* (2010), 291-300.
5. Halverson, C.A., Ellis, J.B., Danis, C., and Kellogg, W.A. Designing task visualizations to support the coordination of work in software development. In *Proceedings of the 20th Anniversary Conference on Computer Supported Cooperative Work* (2006), 39-48.
6. Anvik, J., Hiew, L., and Murphy, G.C. Who should fix this bug? In *Proceedings of the 28th International Conference on Software Engineering* (2006), 361 - 370.
7. Ko, A.J. and Chilana, P.K. How power users help and hinder open bug reporting. In *Proceedings of the 28th International conference on Human Factors in Computing Systems* (2010), 1665-1674.
8. Avnon, Y. and Boggan, S.L. Fit and Finish using a bug tracking system: challenges and recommendations. In *Proceedings of the 28th of the International Conference on Human Factors in Computing Systems (Extended Abstracts)* (2010), 4717-4720.
9. Sandusky, R.J. and Gasser, L. Negotiation and the coordination of information and activity in distributed software problem management. In *Proceedings of the international ACM SIGGROUP Conference on Supporting Group Work* (2005), 187-196.
10. Guo, P.J., Zimmermann, T., Nagappan, N., and Murphy, B. Characterizing and predicting which bugs get fixed: an empirical study of Microsoft Windows. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering* (2010), 495-504.
11. Jeong, G., Kim, S., and Zimmermann, T. Improving bug triage with bug tossing graphs. In *Proceedings of Joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering* (2009), 111-120.
12. Shao, Q., Chen, Y., Tao, S., Yan, X., and Anerousis, N. Efficient ticket routing by resolution sequence mining. In *Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (2008), 605-613.
13. Sandusky, R.J. *Information, activity and social order in distributed work: The case of distributed software problem management*. PhD Thesis, University of Illinois at Urbana-Champaign, 2005.
14. Ripoche, G. and Sansonnet, J.-P. Experiences in Automating the Analysis of Linguistic Interactions for the Study of Distributed Collectives. *Journal Computer Supported Cooperative Work*, 15, 2-3 (June 2006), 149-183.
15. Carstensen, P.H. *Computer Supported Coordination. (PhD Thesis)*. Risø National Laboratory, Roskilde, Denmark, 1996.
16. Aranda, J. and Venolia, G. The secret life of bugs: Going past the errors and omissions in software repositories. In *Proceedings of the 31st International Conference on Software Engineering* (2009), 298-308.
17. van Liere, D.W. How Shallow is a Bug? Open Source Communities as Information Repositories and Solving Software Defects. In *ERIM Report Series Reference Forthcoming. http://ssrn.com/abstract=1507233* (2009).
18. Ko, A.J., Myers, B.A., and Chau, D.H. A Linguistic Analysis of How People Describe Software. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing* (2006), 127-134.
19. Bettenburg, N., Just, S., Schröter, A., Weiss, C., Premraj, R., and Zimmermann, T. What makes a good bug report? In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (2008), 308-318.
20. Just, S., Premraj, R., and Zimmermann, T. Towards the next generation of bug tracking systems. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing* (2008), 82-85.
21. Barker, I. *What is information architecture?*, 2005. KM Column, http://www.steptwo.com.au.
22. Hooimeijer, P. and Weimer, W. Modeling bug report quality. In *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering* (2007), 34-43.
23. Bird, C., Nagappan, N., Devanbu, P.T., Gall, H., and Murphy, B. Does distributed development affect software quality? An empirical case study of Windows Vista. In *Proceedings of the 31st International Conference on Software Engineering* (2009), 518-528.
24. Cohen, J. *Statistical Power Analysis for the Behavioral Sciences*. Routledge Academic, 1988.
25. Hosmer, D.W. and Lemeshow, S. *Applied Logistic Regression*. John Wiley & Sons, 2000.
26. Bird, C., Bachmann, A., Aune, E., Duffy, J., Bernstein, A., Filkov, V., and Devanbu, P.T. Fair and balanced? Bias in bug-fix datasets. In *Proceedings of the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering* (2009), 121-130.
27. Basili, V., Shull, F., and Lanubile, F. Building knowledge through Families of Experiments. *IEEE Trans. Software Eng.*, 25, 4 (1999), 456-473.
28. Anvik, J., Hiew, L., and Murphy, G.C. Coping with an open bug repository. In *Proceedings of the OOPSLA Workshop on Eclipse Technology eXchange* (2005), 35-39.
29. Cataldo, M., Herbsleb, J.D., and Carley, K.M. Socio-technical congruence: a framework for assessing the impact of technical and work dependencies on software development productivity. In *Proceedings of the 2nd ACM-IEEE international symposium on Empirical software engineering and measurement* (2008), 2-11.
30. Cataldo, M., Wagstrom, P.A., Herbsleb, J.D., and Carley, K.M. Identification of coordination requirements: implications for the Design of collaboration and awareness tools. In *Proceedings of the 20th anniversary conference on Computer supported cooperative work* (2006), 353-362.
31. Fritz, T., Ou, J., Murphy, G.C., and Murphy-Hill, E.R. A degree-of-knowledge model to capture source code familiarity. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering* (2010), 385-394.
32. Mockus, A. and Herbsleb, J.D. Expertise browser: a quantitative approach to identifying expertise. In *Proceedings of the 22rd International Conference on Software Engineering* (2002), 503-512.
33. Bragdon, A., Zeleznik, R.C., Reiss, S.P., Karumuri, S., Cheung, W., Kaplan, J., Coleman, C., Adeputra, F., and Jr., J.J.L. Code bubbles: a working set-based interface for code understanding and maintenance. In *Proceedings of the 28th International Conference on Human Factors in Computing Systems* (2010), 2503-2512.
34. Bragdon, A., Reiss, S.P., Zeleznik, R.C., Karumuri, S., Cheung, W., Kaplan, J., Coleman, C., Adeputra, F., and Jr., J.J.L. Code bubbles: rethinking the user interface paradigm of integrated development environments. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering* (2010), 455-464.