MITIGATING THE EFFECTS OF STRUCTURAL COMPLEXITY ON OPEN SOURCE SOFTWARE MAINTENANCE THROUGH ACCOUNTABILITY

Vishal Midha
The University of Texas- Pan American
1201 W. University Dr
Edinburg, TX 78539
vmidha@utpa.edu

Sandra Slaughter
Georgia Institute of Technology
800 W. Peachtree St. NW
Atlanta, GA 30308
sandra.slaughter@mgt.gatech.edu

Abstract

In this research, we investigate the relationships between structural complexity, accountability, and software maintenance performance in Open Source Software development projects. Additionally, we investigate the moderating role of monetary incentives on various relationships. We collected data on 5,000 bug reports from the SourceForge database and perceptual data from 181 open source software developers registered on SourceForge for model validation. Results support our hypotheses. The important implications of the results are discussed.

Keywords: Accountability, Monetary Incentives, Software Maintenance, Open Source Software

Introduction

Typically, an open source project starts when an individual (or group) feels a need for new software or a new feature in existing software to solve a personal or work-related problem, and someone in that group eventually writes the code to meet that need. Once the software is released, the user community can freely utilize it, modify the source code to customize it to their local needs, identify and report errors in the software and submit fixes to existing bugs. Any such code fixes are reviewed by the core group of that open source project, before they are integrated into the source code and released as a new version under the same public license. This process of program refinement and maintenance through bug submission and fixing continues iteratively throughout the useful life of the software. Due to its long-term and iterative nature, open source software (OSS) clearly requires significant long-term investments in maintenance. Software maintenance is the "modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a modified environment" (IEEE 1993). In this study, our focus is on the corrective dimension of software maintenance.

Given the long life of software systems, it is possible that the individual performing maintenance tasks did not participate in the initial development of the original software (Kemerer 1995). If that is the case, the developer has to expend a large portion of his or her resources and mental effort in comprehending the existing software rather than in making the actual modifications (Lientz et al 1978). Maintenance activities for many OSS projects are documented at SourceForge, one of the largest repositories of OSS projects on the Internet. Software testing, code fixing, and potential extensions are often performed by highly skilled experts who invest a considerable amount of their personal time into these projects without expectations of monetary rewards. These open source programmers resemble virtual team members scattered across the globe who rarely meet face-to-face and interact solely via the Internet, but are brought together by their shared interest in software development. Hence, for a volunteer to complete a maintenance task, comprehending the existing source code becomes a key aspect of OSS maintenance. In fact, comprehension may be even more challenging for OSS projects than for traditional proprietary software, given that most of the participants are volunteers who, unlike developers working in companies,

are not bound by any traditional chains of command, and have a choice to not participate in the maintenance activities at all.

As software grows in size and complexity, the task of comprehending it becomes increasingly difficult requiring a greater amount of effort from the developers (Rilling and Klemola 2003, Campbell 1988). However, many psychologists have concluded that individuals are cognitive misers who avoid expending mental effort and attempt to minimize mental procedures whenever possible (Taylor 1981). This might be one reason, among others why developers prefer to work on software development over software maintenance (Polo, Piattini and Ruiz, 2003; Griesser 1993). However, this cognitive miser model was based on laboratory settings; thus, one has to be cautious in assessing the generalizability of this model.

In contrast to laboratory settings, where individuals are not held accountable for the positions they take, in everyday life, individuals operate in settings in which implicit or explicit norms of accountability regulate their conduct (George 1980, Katz and Kahn 1978). When individuals feel accountable for tasks, their task behavior changes (George 1980). The question then inevitably arises: Does accountability influence performance in highly demanding cognitive tasks? More specifically, do individuals perform better in cognitive tasks, such as software maintenance, for which they feel personally accountable?

The concept of accountability, which is defined as an obligation or willingness to accept responsibility or to account for one's actions, has been an inherent part of the design and implementation of information systems. For instance, methods and models such as Yourdon Methodology, Total Quality Methods, and the Capability Maturity Model, focus on ways of making development processes accountable, both within project teams and for management, on an organizational level (Eriksen 2002, Button and Sharrock 1998). Even in the OSS community, the notion of accountability exists through its meritocratic philosophy (Lerner and Tirole, 2002; Masum, 2001; Raymond, 2001). Contributors develop their reputations through sustained quality contributions which lead to recognition and authority in the community. Although accountability has been an inherent part of well-studied and widely-used models of information systems and OSS community development, to the best of our knowledge, its role in software maintenance, and particularly in the OSS setting, has not been studied yet. It is especially important to examine the impact of accountability on software maintenance in OSS projects, because much of the work is done by volunteers. Our research addresses this gap in the maintenance literature by studying the effects of accountability on maintenance performance. In our study, maintenance performance is defined as corrective maintenance effort, or the effort to fix identified bugs in the software.

In addition to software complexity, we also consider the moderating effects of monetary incentives on the relationship between accountability and corrective maintenance effort. Vieder (2008) suggests that monetary incentives confound the impact of accountability on individual's behavior. Clearly, another question then arises: Do monetary incentives moderate the relationship between accountability and performance behavior?

In sum, this paper focuses on formalizing the separate and joint effects of accountability, software complexity, and monetary incentives on performance of corrective maintenance tasks as empirically testable hypotheses. The paper then reports the results from testing those hypotheses within the context of open source software projects using maintenance data from 5,000 bug reports from the SourceForge database and perceptual data from 181 open source software developers registered on SourceForge.

The rest of the paper proceeds as follows. Section 2 presents the design of this study and develops research hypotheses for empirical testing. Research methods employed for testing our hypotheses are described in §3. Data analytic techniques and findings are presented in §4, following by a discussion of these findings in §5. The paper concludes with a summary of its theoretical and practical implications.

Theoretical Framework

The theoretical framework for this study is developed by integrating two different perspectives. The accountability perspective forms the basis for conceptualizing the importance of accountability as motivation to complete software maintenance tasks. The complexity perspective forms the basis for choosing software complexity as a key variable influencing performance.

Accountability Perspective

As observed by Eriksen (2002), accountability is used most frequently by social scientists inspired by ethnomethodology, a research approach which focuses on how people organize their everyday actions and interactions so as to make them visible-and-accountable (Garfinkel 1967). According to Suchman (2002), accountability is bearing the results of one's own actions. It has been argued as a process in which individuals perceive being answerable to external audiences for performing a certain task thereby fulfilling obligations, duties, and expectations (Weigold and Schlenker, 1991, p. 25; London 2003). Using a four stage mode, Schlenker and Weigold (1989) described the processes through which accountability and its resulting effects evolve. The four stages of the model are inquiry, accounting, judgment, and sanctions. In the inquiry stage, individuals anticipate having to describe, interpret, and justify their actions based on their perceived behavioral standards. In doing so, they thereby proffer personal versions of the events and why they occurred; this is the accounting phase. During subsequent stages, individuals' actions are evaluated, and they are rewarded or sanctioned based on a comparison between the behaviors exhibited and behaviors expected. Although the model has four stages, the feeling of accountability and its corresponding behavioral actions take place during the inquiry phase only.

Continuing the work of Schlenker and Weigold (1989), London et al (1997) developed a sub-model of the inquiry phase, which focused only on the components that lead to an individual's feeling of accountability and its corresponding resulting behavior. The model is shown in Figure 1. According to this model, an individual (actor) experiences forces originating due to either internal or external sources for a task (objective). These forces, in turn influence the individual's behaviors and reactions. We have used this model to guide our theoretical foundations for the concept of accountability. The different components of the model are explained next.

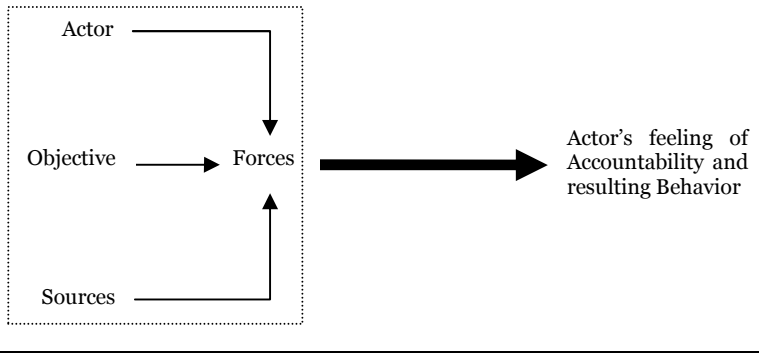

Figure 1: Model of Accountability Processes (Adopted from London et al 1997)

In this model, an individual is held accountable for an objective. An actor's accountability may originate from different sources discussed later. Goal setting theory suggests that the object of accountability has to be clear and unambiguous (Frink and Klimoski, 1998; Locke and Latham, 1990) to ensure that the actor understands what behaviors are expected of him or her. As such, there may be several objectives, and each objective may vary in complexity, and in turn the required effort (Locke and Latham, 1990).

There are two sources of origination of accountability- internal or external. Under external accountability, individuals feel responsible and obligated to perform certain behaviors because they are expected by others or social situations to do so (Erdogan et al, 2004). On the other hand, in the case of internal accountability, the source is one's conscience (London et al, 1997). The individuals, in this case, feel the obligation to perform certain behaviors because of their own commitment to the behavior. Thus, individuals are responding to external accountability when behaviors are performed because of an obligation to another person, and are responding to internal accountability when behaviors are performed as an obligation to oneself. London et al (1997) suggested that accountability may even emerge from a combination of the two sources.

Accountability forces are the reasons why people feel accountable. These forces originate from the internal or external or combination of both sources. Internal forces include feeling of self-control, and ego-gratification. External forces may be financial, approval and recognition from peers, potential positive or negative outcome (e.g. gain in job market) under the discretion of source (London et al 1997). The distinction between the internal and external forces of accountability is important to understand. When the originating source is internal, formal control mechanisms, such as role descriptions, rewards, and punishments, may not be necessary. Instead, individuals carry on the tasks because of their perceived obligation to themselves. However, when the origin is external, the formal mechanisms exert pressure on individuals to perform the expected behaviors. These accountability forces increase an actor's feelings of accountability, which, in turn, influence the actor's behaviors such as increased effort, mindful cognitive processing, faster performance, and satisfaction.

The influence of accountability is currently not very clear, and findings in the literature sometimes appear contradictory. The supporting research suggests that accountability leads subjects to process information in more analytical and complex ways. For example, McAllister et al (1979) found that subjects who felt accountable for their decisions spent more time and effort in a business simulation task than subjects who did not feel accountable for their decisions. Cvetkovitch (1978) argued that accountability led subjects to "more analytical" modes of thought in betting games. In a study on message processing, Tetlock (1983) found that subjects who felt accountable interpreted policy issues on controversial subjects in more multidimensional ways than subjects who did not feel accountable. Chaiken (1980) found that accountable subjects were not influenced by the likableness of the source of the message, but were influenced by the arguments in the message. On the other hand, unaccountable subjects were influenced by the likableness of the source, but not by the arguments. Based on her findings, Chaiken further proposed that accountability led subjects to actively attempt to comprehend and evaluate topic-relevant arguments as opposed to relying on source cues in forming their opinions.

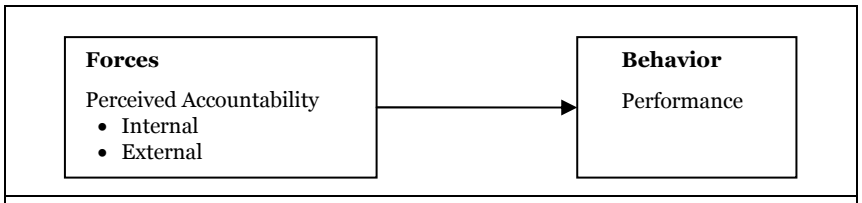

Figure 2: Proposed Model of Accountability

On the other hand, accountability does not always lead to greater cognitive work. A few studies have suggested that accountability simply leads individuals to take positions that they believe are acceptable to others. For example, Adelberg and Batson (1978) found that accountable decision makers in a simulated environment produced less effective results in money allocation activity when there were not enough funds. In an intergroup bargaining experiment, Benton (1972) showed that negotiators who feel accountable to constituents take more rigid bargaining stances and are less likely to arrive at mutually beneficial agreements than are negotiators not under such pressure. Kennedy (1993) found that accountability reduced the recency bias for M.B.A. students, but not for audit managers. Cloyd (1997) found that accountability influenced the search strategies of high-knowledge tax seniors but did not improve the strategies of low-knowledge tax seniors.

The evidence on the effects of accountability is thus mixed: Sometimes accountability leads to complex information processing, and sometimes it leads to expedient decisions. These mixed results could be due to the various differences in studies. For example, the studies by Kennedy (1993) and Cloyd (1997) differed in several aspects such as task complexity, and subjects; and it is possible that these aspects, either alone or in combination, may have accounted for the difference in results. Clearly, further work is needed to understand the impact of accountability. Its impact in the context of software maintenance is developed further in the next section.

Complexity Perspective

Software complexity has existed as an important issue ever since the software programs came into existence. However, there is no consensus about what software complexity actually is. What is accepted is that there are two main categories of software complexity: computational and structural (Zuse 1991). Basili and Hutchens (1983) defines complexity as a measure of the resources expended by a system while interacting with a piece of software to perform a given task. It is important to note the term 'system' in this definition. If the interacting system is a computer, then complexity is defined by the execution time and storage required to perform the computation. For example, as the number of distinct control paths through a program increases, the computational complexity may increase. This kind of complexity is defined as 'Computational Complexity' (Rabin 1977, Curtis et al 1979). If the interacting system is a programmer, then complexity contributes to the difficulty experienced by the programming in performing tasks, such as coding, debugging, testing, or modifying the software. This kind of complexity is known as 'Structural Complexity'. Structural complexity, defined as "the organization of program elements within a program" Gorla and Ramakrishnan, 1997), refers to characteristics of software which make it difficult to understand and work with (Curtis et al 1979). Dealing with structural complexity primarily expends intellectual resources; whereas computational complexity primarily consumes machine resources. As exponential improvements in technology, combined with the ever declining cost of per unit machine resource, have rendered computational complexity unimportant, we primarily focus on structural complexity in this paper.

The notion of structural complexity is linked with the limitations of short term memory. According to the cognitive load theory, all information processed for comprehension must at some time occupy short-term memory (Rilling and Klemola 2003). Short term memory is described as the capacity of information that the brain can hold in an active, highly available state. Short term memory can be thought of as a container, where a small finite number of concepts can be stored. If data are presented in such a way that too many concepts must be associated in order to make a correct decision, then the risk of error increases. The capacity of holding information may vary across individuals, and may limit the capability to comprehend and modify the existing source code.

This structural view of complexity is particularly appropriate for studying software maintenance. Complexity causes two general problems in maintenance. The more complex a system is, the more difficult it is to understand, and therefore to maintain. Empirical studies suggest that a significant portion of the software maintainer's time is devoted to understand the existing source code of the software to be changed (Littman et al 1987). A study of professional maintenance programmers by Fjeldstad and Hamlen (1983), for example, found that, depending on the complexity, the programmers studied the original software code up to 3.5 times as long as they studied the supporting documentation, and equally as long as they spent implementing the modification. The second problem is more insidious. Empirical research shows that complex programs require more maintenance throughout their lives (Gremillion 1984, Vessey and Weber 1983). Complex programs contain more errors, and errors are more difficult to uncover during testing. Consequently, they remain undetected until late in the life cycle. The problem is circular, i.e. maintenance is difficult because of complexity, and because of complexity more maintenance is required. Thus, this study argues that software structural complexity is a key maintenance performance factor because it influences the critical activity of program comprehension.

Research Model and Hypotheses

The theoretical framework for this study is presented in Figure 3. The framework integrates two models in which structural complexity and accountability influence software maintenance performance. These relationships with software maintenance performance are moderated by monetary incentives. A detailed explanation of all the relationships depicted follows.

Maintenance Performance (Time taken to fix bugs)

Prior work on software maintenance has typically measured performance in terms of meeting budget, schedule, and functionality requirements (Hoffer et al 2002). However, OSS projects typically have no a priori budget, schedule, or set of requirements (Scacchi 2002) limiting the conceptualization of effective

development. Similarly, some typical software engineering outcome metrics such as post release defect rates or conformance to quality standards may only be applied toward the end of the systems development life cycle, while attaining effectiveness in earlier stages is important in the OSS context because many projects fail before producing a finished product (Stewart and Gosain 2006).

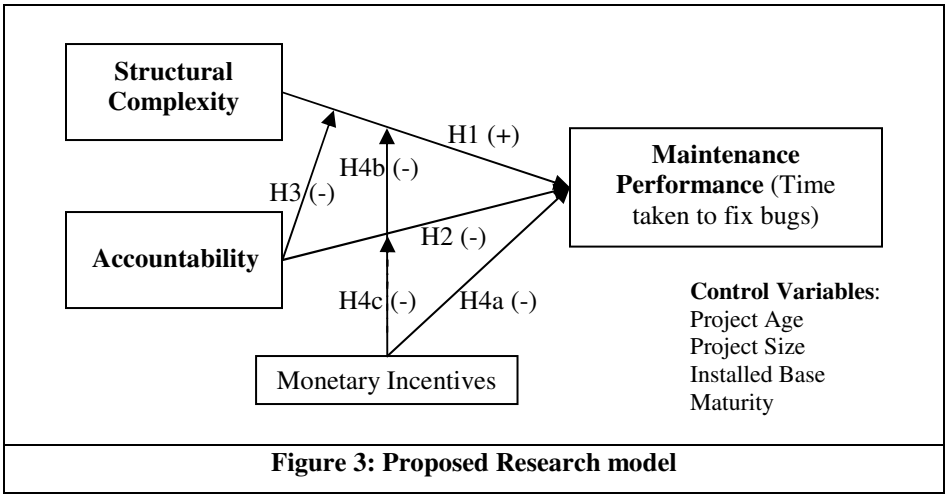

Figure 3: Proposed Research model

Some OSS studies such as Apache and Mozilla have used responses to maintenance requests as indicators of work accomplishment (Mockus et al 2002). Yu (2006) proposed an indirect maintenance model to measure performance of OSS projects in terms of lag time, which is defined as the time between starting a maintenance task and closing the task. In other words, lag time is the duration from when a bug is reported until the bug is fixed and the bug request is closed. Following these, we focus on OSS maintenance performance in terms of the effort (measured as time) to complete corrective software maintenance tasks for OSS projects. Therefore, *Maintenance Performance* is measured as the time logged on SourceForge to complete a corrective maintenance task. The study focuses on the time lag as the variable of interest, since personnel time is the most expensive and scarce resource in software maintenance (Grammas and Klein 1985). In traditional software maintenance, clients are charged by the amount of time spent on their projects. This motivates personnel to maintain accurate project time records.

Complexity

Often a developer fixing bugs in the source code has not participated in the initial development of the original program (Kemerer 1995). As a result, in order to perform maintenance tasks, an individual has to first comprehend the source code of the existing software. Comprehending involves identifying the logic among various segments of the source code and understanding their relationships. This process is essentially a mental pattern recognition by the software developer, and involves filtering and recognizing enormous amount of data (Rilling and Klemola 2003). Consequently, an individual expends most of his or her resources in comprehending the source code rather than in making the actual modifications (Lientz et al 1978).

Kearney et al (1986) suggested that the difficulty of understanding depends, in part, on structural properties of the source code. As simple source code is easier to analyze, it requires less processing capacity. On the other hand, a complex source code is difficult to analyze, and requires high processing capacity. In other words, as software becomes increasingly complex, the amount of required cognitive effort in comprehending existing source code becomes increasingly high (Rilling and Klemola 2003). High cognitive load requires more time-consuming and resource-demanding efforts to familiarize oneself with the code (Darcy et al. 2005). As high amounts of software complexity interfere with the process of comprehending the source code (Gibson and Senn 1989), it becomes difficult for maintainers to efficiently and correctly modify the software. Furthermore, software maintainers are often required to modify

software that is ill-documented, lacks comprehensible structure, and hides data representations (Guimaraes 1983). Thus, software maintenance becomes a largely cognitive task in which programmers perceive and manipulate relationships between informational cues presented by the existing software (Pennington 1987, Shneiderman 1980). This implies that the higher the amount of complexity in source code, the greater the downstream penalties in terms of resources consumed for software maintenance (Banker and Slaughter 2000). Therefore, consistent with the prior research on software complexity, we expect a positive relationship between source code complexity and the time it takes to perform the maintenance tasks.

H1: *An increase in the source code complexity is inversely associated with the software maintenance performance.*

Accountability

Even though the OSS community is open, there is still a notion of accountability through its meritocratic philosophy. The merit is based on reputation gained by recognition from the co-developers and the community members (Lerner and Tirole 2002). When a developer makes a contribution to an OSS project, the peers assess whether the contribution merits acceptance into the code base or not. This acceptance of contribution in the code base brings a "good" reputation to the developer among other co-developers.

Jensen and Scacchi (2007) empirically studied an individual's movement from being an observer to a core developer. They found that an individual moves through different stages – observer, bug reporter, developer, and core developer – while participating in OSS development. During an individual's progression, the individual must both acquire project-specific technical skills, and earn the reputation by demonstrating these skills (Bird et al 2007). Programming skills are considered highly knowledge intensive, and even experienced developers have to work hard to gain specific skills needed for particular development tasks (Bird et al 2007). In other words, an individual earns his reputation by investing a lot of effort, time, and commitment.

When individuals are accountable for a task, whether self-selected or imposed by other members of the OSS community, they attempt to perform the task efficiently to maintain their current status or gain better reputation. Reputation, which emerges as a consequence of the reliable and consistent behavior over time, is an indication of future actions of the individual (Ganesan 1994). Accountability, thus, motivates individuals to perform by increasing the importance of avoiding "bad" standing (embarrassment, loss of self-esteem) and of making "good" standing (status) (Tetlock 1983). In terms of Janis and Mann (1977), people who expect to be evaluated are more likely to perform the tasks. Therefore, an irresponsible action of not completing the task in the timely manner, for which the individuals are accountable, can have negative consequences for their reputation. This fear of loss of reputation motivates a developer to complete the task in timely manner (Markus et al 2000).

Now, when a new bug is reported, OSS project leaders either assign the task of fixing the bug to one particular (set of) developer(s) or do not assign the task to anyone at all. It is also possible that an individual may assign the task to him- or herself. So, when a task is either self-assigned or imposed by project leaders, the individual with the responsibility becomes accountable for that task. In agreement with Fox et al.'s (1979) observation that poor performance of accountable individuals can be highly threatening for their reputation, we argue that the tasks for which someone is accountable, gets completed in shorter time. Hence, we hypothesize that accountability influences an individual's software maintenance performance.

H2: *Accountability is positively associated with software maintenance performance.*

We also argue that accountability moderates the relationship between complexity and maintenance performance. As we have described, software maintenance is a largely cognitive task in which programmers perceive and manipulate relationships between informational cues presented by the existing software (Pennington 1987). High complexity tasks consume more short-term memory and thus require more time-consuming, resource-demanding cognitive-processing of the cues (Darcy et al. 2005). On the other hand, generation of the required output in low-complexity tasks requires only a small proportion of the available cues to be examined and does not require significant effort or resources from

developers (Gibson and Senn 1989). Therefore, accountability, which results in increased mindful cognitive processing and effort, is unlikely to significantly affect the performance of developers for low-complexity tasks. However, such an induced higher cognitive-processing and effort will result in significant improvement of developers' performance in the case of high-complexity tasks, where such effortful information integration is required. Therefore, we hypothesize that accountability will have a stronger impact on the completion time for complex tasks than simple tasks. Hence,

H3: *Accountability will attenuate the relationship between complexity and software maintenance performance.*

Monetary Incentives

Performance suffers if a developer does not wish to exert additional effort or attention, although the capacity to do so exists. A developer will not exert effort unless doing so seems worthwhile. One remedy is to increase the benefits associated with additional attention and effort, perhaps by providing additional stimuli.

Much attention has been given to intrinsic motivational stimuli such as ego gratification, community reputation, and signaling expertise. However, an increasing number of open source projects have opted to receive monetary donations which have been found to have an impact of developers' motivations to participate in OSS development. Commercial firms spent an estimated cumulative Euro 1.2 billion in OSS development up till 2006, both indirectly by allowing or even encouraging their employees to work on public OSS projects or by directly supporting existing OSS (Ghosh 2006). Such monetary incentives provide external stimuli to developers leading them to take deeper interest in the project and devote more of their time and resources to the project (Lakhani and Wolf 2005). Hars and Ou (2002) found salaried and contract programmers to be more strongly motivated by self-determination and personal need. In other words, monetary incentives boost the developers' motivation to input higher levels of effort in maintenance tasks. Similar to the moderating effect in H3, we expect monetary incentives to moderate the relationship between complexity and software maintenance performance. Vieider (2008) argues that as both accountability and monetary incentives trigger increased levels of attention and cognitive processing, confounding the effects of the two makes causal attributions problematic. It may thus be, in principle, that effects traditionally ascribed to *one* are in fact due to the *other*. Thus, the effects should be ascribed to the interaction between monetary incentives and accountability. Based on this suggestion, we propose that accountability and monetary incentives interact to affect software maintenance performance. Therefore, we propose the following:

H4a: *Monetary incentives are positively associated with software maintenance performance.*

H4b: *Monetary incentives will attenuate the relationship between complexity and software maintenance performance.*

H4c: *Monetary incentives will strengthen the effects of accountability on software maintenance performance.*

Methods

We evaluated our hypotheses using data that we collected in two phases. The first phase of data collection involved bug reports that were closed during years 2006 and 2007, whereas the second phase of data collection involved bug reports that were closed between May 2010 and September 2010. The details of the two phases are discussed next.

Phase I

Data were collected from open source projects registered at SourceForge prior to the year 2006. SourceForge is the primary hosting site for open source projects on the Internet, currently hosting many open source projects, and data from this site have previously been used for studying a wide range of open source behaviors (e.g., Grewal et al 2006, von Hippel and von Krogh 2003). Restricting our analysis to projects registered before 2006 allowed us the opportunity to examine at least two years of maintenance

activities for these projects, by examining reported bugs for these projects during the years 2006 and 2007.

Our project sample was further restricted to projects written using the C++ programming language and those designed for the Windows operating system. This was based on prior findings that coders' choice of programming language influences program size (Jones 1986) and program complexity (Weyuker 1988), and hence code written in different programming languages is not directly comparable. Software written in "low" level programming languages tends to have more lines of code and takes longer to understand, correct, or extend than that written in "high" level languages. Likewise, programming efforts tend to vary with the operating system of the project. We chose the Windows operating system because of its widespread deployment and popularity and because of the large number of open source projects directed at this computing platform. This criterion resulted in 1054 projects with 105,910 bugs. For the study, we randomly selected 5,000 closed bugs to test the proposed hypotheses. To ensure reliability of the results, robustness tests were conducted as described in the results section later.

Focal Variables

Maintenance Performance: We extracted various elements of data, including the bugs reported, the date on which the bugs were reported, the date on which the bugs were fixed, and the release number, from bug tracking system, and version tracking reports. From these extracted elements, we calculated the time taken to fix (*FixTime*) each individual bug by subtracting the open date from the close date for each bug.

Complexity: There is a large variety of complexity metrics available in literature and being practiced by the software industry. The two most common metrics are Halstead's E and McCabe's cyclomatic. We view complexity as the degree of cognitive effort involved. From this perspective, Halstead's measures (Halstead 1977) completely ignore the factor corresponding to the complexity of function calls. They are suitable only for predicting algorithmic complexity of a program. If a program has a very small number of operators but many function-calls then these metrics will not give a correct estimation of complexity. In such cases, these metrics will classify an application as a difficult one to comprehend.

McCabe's Cyclomatic complexity is another measure of cognitive complexity. It tends to assess the difficulty faced by the maintainer in order to follow the flow control of the program. It is considered an indicator of the effort needed to understand and test the source code (Stamelos et al 2002). Kemerer and Slaughter (1997) used McCabe's cyclomatic complexity metric to evaluate decision density, which represents the cognitive burden on a programmer in understanding the source code. Therefore, in order to measure the complexity of the task, we measured the cyclomatic complexity of the corresponding version of the project and subjected the source code files through CCCC, a software code analysis tool designed for object-oriented (C++) source code. In the case of C++, the tool counts the number of independent paths using tokens such as 'if', 'while', 'for', 'switch', 'break', '&&', and '||'. The cyclomatic number is calculated once for each function, wherever it appears in the inheritance hierarchy, and is considered to be contributed exactly once to the total cyclomatic number for the system as a whole, no matter how many child classes gain access to the function by inheritance. To account for the effects of size, the complexity (*Complexity*) was normalized by dividing it by the number of lines of code for each software project. In addition, this also reduces collinearity problems when size is included in the regression models (Gill and Kemerer 1991).

Accountability: Each project in our study maintains a list of tasks to be completed. For each task, projects also list the individual to whom the task has been assigned (either by the group administrators or chosen by the individual). If the task is not assigned to any one, it is left open for so that anyone from the OSS community can work on it. Accordingly, a variable *Accountability* is used to represent task accountability (1 if assigned, and 0 otherwise).

Monetary Incentives: A control variable *Sponsor* is used as a dummy variable to denote whether a project uses external funds as part of its incentive mechanism (1 if yes, and 0 otherwise).

Control Variables

Different variables including *age* of project, *size* of project, total *installed base* representing total downloads, and *project maturity* were controlled for. Detailed explanations for controlling these variables have been omitted due to page length limitations but are available upon request from the authors.

Selection Bias

As a significant portion of the bugs (34%) were not closed within the time frame of the study, the selection of only closed bugs could have resulted in selection bias in the results. In order to test for the possibility of sample selection bias, we compared various chacteristics of bugs that were closed (and selected in the study) with the chacteristics of bugs that were not closed (not selected in the study). We randomly sampled 100 bugs that were not closed, and extracted data elements for complexity, project size, and sponsorship for those bugs. The results of all the t-tests on the two groups show no significant differences. This suggests that selection bias may not be salient.

Analysis Technique

Although we have controlled for various project specific characteristics that could affect time to fix bugs, there could still be potential unobserved heterogeneity. Unobserved attributes such as governance mechanisms, coordination among the developers, etc., may potentially influence maintenance performance. The presence of such unobserved effects may lead to inefficient estimates. Hence, to check for the significance of these unobserved effects, we perform a Breusch-Pagan Lagrange Multiplier test (Wooldridge 2006). In the absence of project specific unobserved effects a pooled OLS estimation is consistent as well as efficient. In the presence of unobserved effects but the absence of correlation between unobserved and observed independent variables random effects estimation is consistent as well as efficient. However, in the presence of the correlation the random effects estimation is inconsistent. Given the potential for correlation between the unobserved characteristics of a project and the observed independent variables, we test these effects through the use of project fixed effects estimation (Greene 2003). By the use of fixed effects, we can parcel out the effect of project level factors and obtain unbiased estimates of the relationship between independent and dependent variables. Though the fixed effects estimates are consistent, they are inefficient in the absence of correlation. We performed the Hausman specification test to compare fixed versus random effects under the null hypothesis that the project specific unobserved effects are uncorrelated with other regressors in the model (Greene 2003). The results are presented in a subsequent section.

Phase II

As discussed in the theory development section, accountability forces are the reasons why individuals feel responsible for performing certain tasks. Such feelings of accountability can originate from internal as well as external sources. To capture accountability as a perceptual force, we also measured perceptions of accountability using a survey instrument. The instrument was adopted from Frink and Ferris (1998), and contained 5 items each for internal and external accountability. The instruments included items such as, "I will get recognition from the project administrators if I fix this bug," "My success in fixing this bug is important to the project administrators," "The project administrators feel that fixing this bug is important for the project," "The project administrators' impression of how quickly I fixed this bug is important to me," "I feel accountable to the project administrators to fix this bug."

An important section of the survey involved a retrospective study, where the respondents were asked to recall a specific bug they fixed, and their perceptions at the time of fixing that bug. To avoid memory bias, we chose to limit the data collection to bugs reports closed within the last 4 months, i.e., May 2010 and September 2010. The other selection criteria were kept the same as discussed in Phase 1. Following the

preparation of a list of bug reports between the aforementioned time period, 300 email invites were sent to individuals who had fixed those respective bugs, with the instructions to answer the survey based on the respective bugs. Each invitation and corresponding survey was customized to include the web link to specified bug report.

Of the 300 emails sent, we received 181 usable responses (60.33% response rate). A t-test was performed to compare the experience (length of registration at SourceForge) of developers who responded to the survey and the developers who did not respond to the survey. A separate t-test was performed on the number of the projects with which the developers were associated. Both t-tests confirmed no differences in the responders and non-responders. Hence, non-response bias was not an issue. Subsequent to receiving the responses, we gathered other data items as well as objective measures of accountability for each bug as discussed earlier in Phase 1.

Phase II of our study serves two purposes: (i) validates our objective measure of accountability, and (ii) confirms the reliability of our model and results.

Model Specification and Results

Initial investigations indicated that the dependent variable and many of the independent variables were not normally distributed. In such cases, linear regression analysis might yield biased and non-interpretable parameter estimates (Gelman and Hill 2007). Therefore, as suggested by Gelman and Hill (2007), logarithmic transformations were performed on the non-normally distributed dependent and independent variables.

Additionally, interaction terms were found to be correlated with their corresponding variables. The correlations were high enough to raise concerns of multicollinearity, which if uncorrected for may lead to inflated standard errors and, in worst case, inconsistent or unstable estimates (Greene, 2003). As suggested by Gelman and Hill (2007), we mean centered the variables before calculating the interaction variables which reduced the correlation to acceptable level. Multicollinearity does not appear as an issue. For clarity, the different transformations are shown as ln, nm, and mc for log operated, normalized, and mean centered transformations respectively in the model.

For our dependent measure, *FixTime*, we test the impact of complexity and accountability by estimating the parameters for the following regression model:

$$lnFixTime_{ij} = \alpha + \beta_1 nmComplexity_{ij} + \beta_2 Accountability_{ij} + \beta_3 mcAccountability_{ij} * mcComplexity_{ij} + \beta_{4a} mcSponsor_i + \beta_{4b} mcSponsor_i * mcComplexity_{ij} + \beta_{4c} mcAponsor_i * mcAccountability_{ij} + \beta_5 lnAge_{ij} + \beta_6 lnSize_{ij} + \beta_7 lnDownload_{ij} + \beta_8 Maturity_{ij} + \mu_j + \varepsilon_{ij}$$

where *i* is project; *j* is bug; lnFixTime is log of dependent variable (time taken to fix the bug_{ij}); $Complexity_{ij}$ is the complexity of the project at the time of the bug fix; $Accountability_{ij}$ is the indicator variable which equals 1 when someone is accountable for the bug fix maintenance task, and 0 otherwise; $Sponsor_j$ is an indicator variable which equals 1 when a project receives monetary sponsorship, and 0 otherwise; $Priority_{ij}$ is also an indicator variable which equals 1 when the bug fixing task has high importance, and 0 otherwise; u_i represent unobserved effects; and ε_{ij} is the error term.

Results for Phase I

The Breusch-Pagan Lagrange Multiplier test was significant, (χ^2= 6.41, $p < 0.05$) and the Hausman test was insignificant (χ^2 = 10.50, p=n.s.). This implies that the random effects estimation is consistent as well as efficient for the analyses. The residuals of the all models were analyzed for potential serial correlation. No significant serial correlation was observed. As a further diagnostic test, we calculate Variance Inflation Factors (VIF) for each variable (Greene 2003); the maximum value of VIF is 1.40 (see Table 1), well below 10, which is usually considered as the threshold above which multicollinearity may affect results (Neter et al. 1990). Belsley-Kuh-Welsch (1980) collinearity diagnostics indicated a highest condition number for

the model of 10.37, also within the recommended limit (Greene 1993), confirming no significant concern for multicollinearity. A variety of specification checks was performed for the estimated model to ensure that standard assumptions were satisfied. Table 1 presents random effects estimation results for the time it takes to close bugs. The results are based on data collected for 5,000 bugs from 552 different OSS projects.

The results show support for all six proposed hypotheses. The coefficient of complexity is positive and significant supporting hypothesis 1. Consistent with hypothesis 2, the coefficient of Accountability is negative and significant. The interaction between Complexity and Accountability is negative and significant supporting hypothesis 3. Consistent with hypothesis 4, the coefficient for monetary incentives is negative and significant. Monetary incentives variable also has significant moderating effect on both accountability and complexity. The results are discussed in details in the next section.

Robustness Test

Other than applying robust standard errors and testing for multicollinearity in Table 1, we further performed sensitivity analyses included re-estimating maintenance performance after deleting two influential observations identified using the Belsley-Kuh-Welsch (1980) criteria. The sign and significance of the variables in the revised regression correspond to those in the original model. Additionally, we computed the regression coefficients using a fixed effects model, and the regression results were consistent with the results from the random effects model. To conserve space we do not report these results. The results are available from the authors upon request.

Table 1: Results of Random Effects Models for lnFixTime (n=552, Obs=5000)

Variable	Coefficients	Std Error	VIF
Focal Effects			
Complexity	.472**	.012	1.40
Accountability	-.373**	.023	1.14
Complexity*Accountability	-.152**	.028	1.33
Sponsor	-.347**	.027	1.37
Complexity*Sponsor	-.211**	.029	1.17
Accountability* Sponsor	-.277**	.051	1.30
Control Effects			
Project Age	.038	.026	1.16
Project Size	.005	.006	1.04
Installed Base	-.016*	.005	1.03
Project Maturity	.006	.005	1.01
Model Statistics			
R-square within	.347		
R-square between	.461		
R-square overall	.356		* p-val < 0.05
Wald $\chi 2$	3624.59		** p-val < 0.01

NOTE: The dependent variable was computed in terms of time taken to close bugs. Therefore, the higher value would imply lower maintenance performance, whereas lower value implies higher maintenance performance. Consequently, the estimated regression coefficients have reverse signs.

Results for Phase II

The measurement scale for accountability was tested for reliability and construct validity using confirmatory factor analysis (CFA). CFA is more appropriate than alternative statistical techniques such as exploratory factor analysis when there is strong a priori theory and the research employs mostly pre-validated measurement scales (Bagozzi and Phillips 1982), as was the case in this study. Confirmatory factor analysis utilizing varimax rotation (Johnson and Wichern 1992) and the Kaiser-Meyer-Olkin (1974) measure of sampling adequacy and Bartlett's test of sphericity were employed to verify that internal and external accountability constructs loaded on orthogonal factors. The results of factor analysis as shown in Table 2 confirmed a two-factor solution.

Table 2: Confirmatory Factor Analysis & Scale Properties

Internal		External	
1a	.833	2a	.887
1b	.887	2b	.882
1c	.869	2c	.893
1d	.836	2d	.839
1e	.823	2e	.867
Composite Reliability	.929	Composite Reliability	.941
AVE sq root	.850	AVE sq root	.874

The convergent validity of scale items was assessed using three criteria suggested by Fornell and Larcker (1981): (1) all item factor loadings (λ) should be significant and exceed 0.60, (2) composite reliabilities for each construct should exceed 0.80, and (3) average variance extracted (AVE) for each construct should exceed 0.50, or in other words, the square root of AVE should exceed 0.71. As seen from Table 2, standardized CFA loadings for all scale items in the CFA model were significant at $p < 0.001$ and exceeded the minimum loading criterion of 0.60, with the minimum loading being 0.823 for internal accountability item 1e. From Table 2, we can see that composite reliabilities of all factors also exceeded the required minimum of 0.80, with the values of being 0.929 and 0.941 for internal and external accountability constructs respectively. Further, the square roots for AVE for both the constructs are 0.85 and 0.87; both of these are greater than the desired minimum of 0.71. Hence, all three conditions for convergent validity were met.

Table 3 presents random effects estimation results for the time it takes to close bugs. The results are based on data collected for 181 bugs from 159 different OSS projects. Model 2a shows the results of random effects estimation for Phase II data with an objective measure of accountability, whereas Model 2b shows the results for data with perceptual measure (through survey) of accountability. Both models show support for all six proposed hypotheses. More importantly, the beta coefficients in both the models have similar values indicating that the two measures have similar effects. This is also confirmed through the high value of coefficient of concurrent validity (0.716) between the two measures of accountability.

Model 2c and 2d show the results of post-hoc analysis, where we tested the effects of accountability originating from internal and external forces separately. Model 2c shows the results of random effects estimation with accountability originating from internal sources, whereas Model 2d shows results with accountability originating from external sources. As can be seen in Model 2c, the direct effect of internal accountability on the maintenance performance is not statistically significant at the 0.05 level.

Discussion

This study employed data from open source software projects to analyze the impact of complexity and accountability on software maintenance performance (defined in terms of the time taken to complete

corrective maintenance tasks). Various other moderating effects were also studied. This section describes our key findings and their implications for software maintenance research and practice. Unless mentioned otherwise, the discussion is based on the results of Phase I data.

Complexity can lead to many problems, an obvious one being the time taken to fix bugs. It is common that when a bug is fixed in one segment of the source code, it usually causes ripple effects and adjustments in other segments. The more complex the software, the more the required adjustments in other segments. As a consequence, the developer has to simultaneously understand, and repair related pieces in dispersed segments. Handling all segments together has a detrimental effect on the time devoted by the developer because more time is needed to follow the flow of logic within the code. We found that, ceteris paribus, with an increase in one unit of complexity, the average time it takes to fix bugs increases by 0.472 units.

Table 3: Results of Random Effects Models for lnFixTime (n=159, Obs=181)

	Model 2a (0/1 Scale)	Model 2b (Overall Acct)	Model 2c (Int. Acct)	Model 2d (Ext. Acct)
Focal Effects				
Complexity	0.317**	0.355**	0.354**	0.363**
Accountability	-0.203**	-0.186**	-0.108	-0.153*
Complexity*Accountability	-0.194*	-0.160*	-0.168*	-0.126*
Sponsor	-0.129*	-0.163*	-0.228**	-0.242**
Complexity*Sponsor	-0.149*	-0.172*	-0.214**	-0.203*
Accountability* Sponsor	-0.160*	-0.145*	-0.144*	-0.039
Control Effects				
Project Age	0.092	0.096	0.099	0.096
Project Size	0.041	-0.002	-0.007	-0.001
Installed Base	-0.046	-0.044	-0.046	-0.056
Project Maturity	-0.058	-0.034	-0.041	-0.031
Model Statistics				
R-square within	0.220	0.172	0.134	0.194
R-square between	0.481	0.482	0.476	0.451
R-square overall	0.443	0.432	0.422	0.416
Wald $\chi 2$	294.720	262.930	270.180	269.200

* p-val < 0.05; ** p-val < 0.01

Model 2a: Direct measure of accountability (0/1 scale as in Phase I);
Model 2b: Accountability measured using overall perceptual measure (0-5 scale)
Model 2c: Accountability measured using only Internal perceptual measure (0-5 scale)
Model 2d: Accountability measured using only External perceptual measure (0-5 scale)

This result has another spurious effect on software maintenance. When a developer becomes conscious of the long time needed to fix a bug, there is a tendency for the developer find 'quick and dirty' solutions, thereby making the code even less maintainable. Such half-baked efforts lead to a vicious cycle in which the complexity, the number of bugs, and the time taken to fix those bugs feed on each other until a dead end is reached with the only option of either reengineering the project or shutting it down completely.

According to the theoretical foundations, accountability as liability rests both on causation of effects and on the social expectations of others for one's action. When held accountable for a task, an individual attempts to avoid negative consequences and attempts to achieve positive consequences. Based on this, Tetlock (1983) suggested accountability as a technique to improve performance. Our results concur with

the theoretical model. We found that with an increase in one unit of accountability, the average time taken to fix bugs decreases by 0.373 units.

Apache Group, the informal organization of people responsible for guiding the development of the Apache HTTP Server Project, allocates the accountability for bug fixes to its developers, and has been able to get work done in an effective way. Project leaders, however, must be aware that they allocate the accountability of the work to someone who has the capabilities and resources to do it. Allocating a task to a developer, who does not possess sufficient required knowledge or resources, may lower the developer's activity and increase the average time it takes to complete the task.

Even though complexity and accountability have significant impact on maintenance effort, in order to understand their true effects, it is important to consider the moderating effects. The total impact of complexity on the time to fix bugs is 0.472*Complexity – 0.152*Complexity*Accountability – 0.211*Complexity*Sponsor. So the coefficient 0.472 represents the full effect of a one unit change in Complexity when holding everything else constant and Accountability*Sponsor=0. However, the full effect of one unit change in Complexity when Accountability and Sponsor are not zero is 0.472 – 0.152*Accountability – 0.211*Sponsor holding all other variables constant. The actual impact of Complexity is 0.374 in the presence of Accountability and Sponsor (computed at their respective mean values). Note that the effect of one unit change in Complexity is decreasing in the value of Accountability and Sponsor. The significant moderating impact of accountability on the relationship between complexity and performance is that accountability motivates effort-demanding information processing, i.e. individuals exert more effort on complex tasks to maintain their current or gain reputation. This implies that delegating accountability of tasks to a developer will lead to better performance of highly complex software maintenance tasks. The results for the moderating effects for complexity are shown in Fig 2. The results for the impact of accountability in the presence of accountability also showed similar trends.

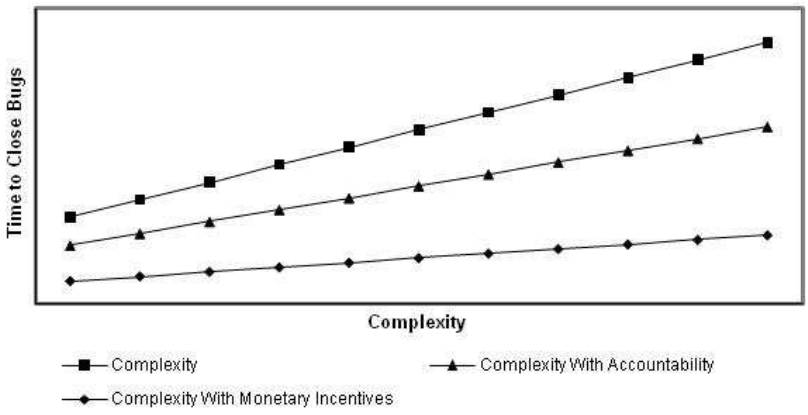

Figure 2: Effect of Complexity with Accountability and Monetary Incentives

The true effect of accountability on software maintenance is -0.373*Accountability – 0.277* Accountability* Sponsor. So the coefficient -0.373 represents the full effect of a one unit change in Accountability when holding everything else constant and Sponsor=0. However, the full effect of one unit change in Accountability when Sponsor is not zero is -0.373 – 0.277*Sponsor holding all other variables constant. The actual impact of Accountability is -0.427 in the presence of Sponsor (computed at mean value). Clearly, the decrease in time to close bugs with increase in accountability was significantly larger for the projects that received monetary incentives compared to the projects that did not receive any monetary incentives. The results are shown in Fig 3 and suggest that developers take accountability more seriously when monetary incentives are attached with a project.

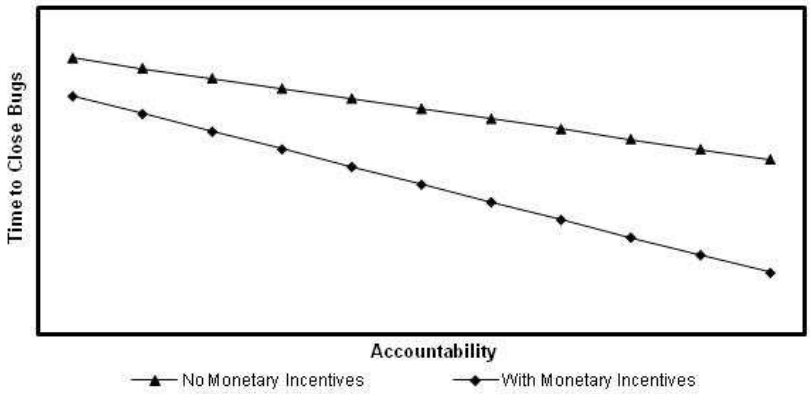

Figure 3: Effect of Accountability with Monetary Incentives

The results of the Phase II study also show similar findings. One particular result is worth separate discussion. As noted, the effect of accountability when the originating source was internal was not found to be significant. However, the effect was significant when the source was external (Model 2d). The results imply that the external forces of accountability have a stronger impact on maintenance performance than the internal forces. The subtle but critical difference between external accountability and internal accountability is "the boss holds me accountable" and "I hold myself accountable". Katzenbach and Smith (1993) point out that external accountability is a precursor to internal accountability, i.e., external accountability helps in developing a stronger and a more powerful sense of internal accountability. At its core, accountability is about the promises that underpin the two critical aspects of effective and mature teams: commitment and trust (Katzenbach and Smith 1993). They further suggest that accountability arises from and reinforces the time, energy, and action invested in figuring what the team is trying to accomplish. As our results find only external accountability to be significant, the implication is that OSS communities are still in the early stages of mature teams. At this time, individuals in the OSS communities in our study appear to hold themselves accountable externally, and may be developing the ability to hold themselves accountable internally. However, a full-fledged separate study in future might be needed to attest to this.

Limitations

Like most empirical studies, this research is not without limitations. The first limitation is the sample frame. While Sourceforge has data about many open source projects, it does not capture all open source projects, which was the population of interest in this study. However, the selected sample size was large enough to ensure statistical validity.

Second, findings from our sample of open source software may not be entirely generalizable to software maintenance projects (including proprietary) at large. For instance, it is possible that some bugs may not be reported in open source projects, given the voluntary participative nature of such projects, which may influence some of our findings.

Third, sometimes tasks are not assigned to someone right away. For example, a task may be left open now, but gets assigned to a developer next week. One could also argue that an individual may become accountable (self-imposed) for a task after working on it, and then submit the modification immediately to achieve higher reputation. We did not differentiate between such tasks as the results will remain the same because we measured lag time between the date on which a bug was reported and the date on which the bug was closed. Even if the developer submits the modification immediately, the developer would have worked on it in order to submit the modification. However, controlling for the date of task delegation may produce interesting results, and we leave that as an option for future research.

Implications

In spite of the above limitations, this study has important implications for research and practice. To the best of our knowledge, this is the first study to investigate the impact of accountability on software maintenance performance. Strong empirical support for our hypothesized relationships suggests that accountability is indeed useful for software maintenance projects, including open source projects that tend to have a less rigorously controlled development structure compared to proprietary software projects.

These findings have at least two immediate implications for software and project managers. First, they must monitor and control complexity in order to achieve efficient software maintenance. Second, they must delegate accountability to improve software maintenance performance. Maintenance tasks are completed in less time when the accountability is assigned. And third, project managers should solicit more external incentives for programmers to work on projects.

Finally, although we examined the time taken to complete corrective maintenance tasks as a measure of maintenance performance, the time taken to fix a bug is also indicative of software quality. Well-designed software should theoretically require less time to correct bugs. Hence, it is possible that accountability may also help improve software quality. We leave such analysis as an option for future research.

References

Adelberg, S., and Batson, C. D. 1978. "Accountability and helping: When needs exceed resources". *Journal of Personality and Social Psychology*, 36, 343-350.

Arthur, L. J. 1988. *Software Evolution*, John Wiley & Sons, Inc., New York.

Basili and Hutchens, 1983. "An Empirical Study of a Syntactic Complexity Family, IEEE Transactions on Software Engineering, (9, 6), p.664-672.

Belsley, D. A., E. Kuh, and R. E. 1980. Welsch, Regression Diagnostics: Identifying Influential Data and Sources of Collinearity, Wiley, NewYork.

Benton, A. A. 1972. "Accountability and negotiations between group representatives". *Proceedings of the 80th Annual Convention of the American Psychological Association*, 7, 227-228.

Bird, C., A Gourley, P Devanbu, A Swaminathan, G Hsu. 2007. "Open Borders? Immigration in Open Source Projects". *Proceedings of the 4th International Workshop on Mining Software Repositories.*

Button, G. and Sharrock, W. 1998. "The Organizational Accountability of Technological Work". *Social Studies of Science* (28, 1), 73-102.

Chaiken, S. 1980. "Heuristic versus systematic information processing and the use of source versus message cues in persuasion". *Journal of Personality and Social Psychology*. 39, 752-766.

Cloyd, C. B. 1977. "Performance in Tax Research Tasks: The Joint Effects of Knowledge and Accountability." *The Accounting Review,* 111-31.

Curtis, B., Sheppard, S., Milliman, P., Borst, M., and Love, T. 1979. "Measuring the Psychological Complexity of Software Maintenance Tasks with the Halstead and McCabe Metrics," *IEEE Transactions on Software Engineering*, 5(2), pp.96-104.

Cvetkovitch, G. 1978. "Cognitive accommodation, language, and social responsibility". *Social Psychology*, 41, 149-55.

Darcy, D., Kemerer, C., Slaughter, S., and Tomayko, J. 2005. "The Structural Complexity of Software: An Experimental Test," *IEEE Transactions on Software Engineering*, 31(11), pp.982-995.

Dvorak, J. "Conceptual entropy and its effect on class hierarchies". IEEE Computer 27(6): 59-63, 1994.

Fenton, N. E. and Pfleeger, S. L. Software Metrics: A Rigorous and Practical Approach, Revised, Boston, MA: International Thomson Publishing, 1997.

Fjeldstad, R. K. and W. T. Hamlen, "Application Program Mainte- nance Study-Report to our Respondents," in G. Parikh and H. Zvegintzov (Eds.), Tutorial on Software Maintenance, IEEE Computer Society Press, Silver Spring, MD, 1983

Fox, F. V., and Staw, B. M. 1979. The trapped administrator: The effects of job insecurity and policy resistance upon commitment to a course of action. *Administrative Science Quarterly*, 24, 449—471.

Ganesan, S. 1994. "Determinants of long-term orientation in buyer–seller relationships", *Journal of Marketing* 58, pp. 1–19.

Garfinkel, H. 1967. *Studies in Ethnomethodology*. Polity Press/ Blackwell Publishers, Oxford, UK, 1984. [First published in USA, 1967.]
Gelman, A., and Hill, J. 2007. *Data Analysis Using Regression and Multilevel/Hierarchical Models*, Cambridge University Press.
George, A. 1980. *Presidential decisionmaking in foreign policy: The effective use of information and advice*. Boulder, Colo,: Westview Press.
Ghosh 2006. http://stuermer.ch/blog/documents/FLOSSImpactOnEU.pdf
Gibson, V. R. and J. A. Senn, 1989. "System Structure and Software Maintenance Performance," *Comm. ACM*, 32, 3, 347–358.
Gill, G. and Kemerer, C. 1991. "Cyclomatic complexity density and software maintenance productivity," *Transactions on Software Engineering*, 17(12), 1991, pp.1284–1288.
Gorla, N. and R. Ramakrishnan, 1997. "Effect of software structure attributes on software development productivity". *Journal of Systems and Software*, 36(2): p. 191-199
Grammas, G. W., and Klein, J. R. 1985. "Software Productivity as a Strategic Variable." *Interfaces*. (15, 3), pp. 116-126.
Greene, W.H. 2007. *Econometric Analysis*, 6th Edition, Prentice Hall.
Gremillion, L.L. (1984), Determinants of Program Repair Maintenance Requirements,Communications of the ACM 27, 8, 826–832.
Grewal, R., Lilien, G., Mallapragada, G. "Location, Location, Location: How Network Embeddedness Affects Project Success in Open Source Systems," Management Science 52(7), 2006, pp.1043–1056.
Griesser, J.W., (1993) "Motivation and Information System Professionals", Journal of Managerial Psychology, Vol. 8 Iss: 3, pp.21 – 30
Guimaraes, T., 1983. "Managing Application Program Maintenance Expenditures," *Comm. ACM*, 26, 10, 739–746.
Halstead, M.H. Elements of Software Science. Elsevier North Holland, 1977
Hoffer, J. A., George, J. F., and Valacich, J. S. Modern Systems Analysis and Design (Third ed.). New Jersey: Prentice Hall. 2002.
IEEE 1993. IEEE Std. 1219: Standard for Software Maintenance. Los Alamitos CA., USA. IEEE Computer Society Press.
Janis, I. L., and Mann, L. *Decision making*. New York: Free Press, 1977.
Jensen, C., and Scacchi, W. (2007). Role Migration and Advancement Processes in OSSD Projects: A Comparative Case Study. Paper presented at the 29th International Conference on Software Engineering (ICSE)
Katz, D., and Kahn, R. L. The social psychology of organizations (2nd ed.). New York: Wiley, 1978
Katzenbach, Jon R. and Douglas K. Smith (1993), "The Discipline of Teams," Harvard Business Review, 71 (March/April), 111-20.
Kearney, J., Sedlmeyer, R., Thompson, W., Gray, M., and Adler, M. "Software Complexity Measurement," Communications of the ACM, 29(11), 1986, pp.1044-1050.
Kemerer, C. and. Slaughter, S. 1997."Determinants Of Software Maintenance Profiles: An Empirical Investigation," *Software Maintenance: Research And Practice*, 9(4), pp.235–251.
Kemerer, C. F. "Software complexity and software maintenance: A survey of empirical research," Annals of Software Engineering, 1(1), 1995,pp.1-22.
Kemerer, C.F., and Slaughter, S. (1999). An Empirical Approach to Studying Software Evolution, IEEE Trans. Softw. Eng., 25(4), Jul./Aug. 99, pp. 493-509
Kennedy, J. (1993). Debiasing audit judgment with accountability: A framework and experimental results. Journal of Accounting Research, 31, 231-245.
Lakhani, K., and Wolf, B. "Why Hackers Do What They Do: Understanding Motivation and Effort in Free/Open Source Software Projects," Perspectives on Free and Open Source Software, MIT Press, Cambridge, 2005.
Lawton, G. (2002). The great giveaway. New Scientist. http://www.newscientist.com/article/mg 17323284.600-the-great-giveaway.html. Accessed Nov 22, 2009
Lerner, J., and Tirole, J. 2000. The Simple Economics of Open Source," NBER Working Paper 7600, The National Bureau of Economic Research, Inc., Cambridge, MA.
Lientz, B. P., Swanson, E. B., and Tompkins, G. E. 1978. "Characteristics of Application Software Maintenance," *Communications of the ACM* (21:6), pp. 461-471.
Littman, D. C., J. Pinto, S. Letovsky, and E. Soloway, 1987."Mental Models and Software Maintenance," *J. Systems and Software*, 7, 341-355.

Markus, M. L., Manville, B., and Agres, C. E. 2000. "What Makes a Virtual Organization Work?," *Sloan Management Review*, pp. 13-26.

Masum, H. 2001. "Reputation layers for open source development. In: Making Sense of the Bazaar": *Proceedings of the 1st Workshop on Open Source Software Engineering*. Feller, J., Fitzgerald, B. and van der Hoek, A. (eds). http://opensource.ucc.ie/icse2001/papers.htm.

McAllister, D. W, Mitchell, T. R., and Beach, L. R. The contingency model for the selection of decision strategies: An empirical test of the effects of significance, accountability, and reversibility. Organizational Behavior and Human Performance, 1979, 24, 228-244.

Mockus, A., Fielding, R. T., and Herbsleb, J. D. "Two Case Studies of Open Source Software Development: Apache and Mozilla," ACM Transactions on Software Engineering and Methodology (11:3), 2002, pp. 309-346.

Neter, J., W. Wasserman, and M. H. Kutner, Applied Linear Statistical Models, 3rd ed., Irwin, Homewood, IL, 1990.

Pennington, N., "Stimulus Structures and Mental Representations in Expert Comprehension of Computer Programs," Cognitive Psychology, 19 (1987), 295–341.

Pigoski, T. Practical Software Maintenance. Wiley computer publishing, 1997

Polo, M., M. Piattini, and F. Ruiz, editors. Advances in software maintenance management: Technologies and solutions. Idea Group Publishing, U.S.A, 2003.

Rabin, M. O. Complexity of computations, Communications of the ACM, v.20 n.9, p.625-633, Sept. 1977

Raymond, E. The cathedral and the bazaar: musings on Linux and open source by an accidental revolutionary, Sebastopol,CA, O'Reilly, 2001.

Rilling, J. and Klemola, T. "Identifying Comprehension Bottlenecks Using Program Slicing and Cognitive Complexity Metrics," Proceedings of the 11th IEEE International Workshop on Program Comprehension, 2003, pp.115.

Scacchi, W. "Understanding the Requirements for Developing Open Source Software Systems," IEE Proceedings on Software (149:1), 2002, pp. 24-39.

Shneiderman, B., Software Psychology: Human Factors in Computer and Information Systems, Winthrop Publishers, Inc., Cambridge, MA, 1980.

Stamelos, I.; Angelis, L.; Oikonomou, A.; and Bleris, G. "Code Quality Analysis in Open Source Software Development," Information Systems Journal, 12(1), 2002, pp.43-60.

Stewart, K. and Gosain, S. 2006. "The impact of ideology on effectiveness in open source software development teams". *MIS Quarterly* 30, 2, 291–314.

Suchman, L. 2002. "Located accountabilities in technology production". *Scand. J. Inf. Syst.* 14, 2, 91-105.

Taylor, S. E. 1981. "The interface of cognitive and social psychology". In J. Harvey (Ed.), *Cognition, social behavior, and the environment* (pp. 182–211). Hillsdale, NJ: Erlbaum

Tetlock, P. E. 1983. "Accountability and complexity of thought". *Journal of Personality and Social Psychology*, 45, 74-83.

Vessey, I. and R. Weber 1983. "Some Factors Affecting Program Repair Maintenance: An Empirical Study," *Communications of the ACM* 26, 2, 128–134.

Vieider, F. M. 2011. "Separating Real Incentives and Accountability". *Experimental Economics* (Forthcoming)

von Hippel, E., G. von Krogh. 2003. "Open Source Software and the "Private-Collective" Innovation Model: Issues for Organization Science," *Organization Science*, 14(2), pp.209–225.

Weyuker, E. 1988. "Evaluating software complexity measures," *IEEE Transactions on Software Engineering*, 14(9), pp.1357-1365.

Zuse, H. 1991. Software complexity: Measures and methods. deGruyter, NY.