# Effective Corrective Maintenance Strategies for

# Managing Volatile Software Applications

Chris F. Kemerer

University of Pittsburgh

Gregory D. Moody

University of Nevada, Las Vegas

Sandra A. Slaughter

Georgia Institute of Technology

<acknowledgements will go here>

# Effective Corrective Maintenance Strategies for

# Managing Volatile Software Applications

<author names and affiliations redacted>

# Effective Corrective Maintenance Strategies for Managing Volatile Software Applications

## Abstract

### Corrective software maintenance is a difficult knowledge-based task, as maintainers must understand the software and its context prior to performing any maintenance. Managers rely upon technologies, experience and/or skilled developers to assist in the conveyance of knowledge between maintainers and knowledge sources. In this study we theorize how software volatility (measured as the frequency, predictability and magnitude of maintenance change events) alters the effectiveness of these three approaches to knowledge sharing. We test our hypotheses using data from a rare three-year longitudinal panel dataset. When applications are modified less frequently, the technology-based approach results in fewer errors, whereas the experience- and skill-based approaches are more effective when applications are frequently modified. Unpredictable modifications enhance the effectiveness of skill-based approaches, and larger modifications enhance the effectiveness of technology- and skill-based approaches. These results can aid managers in better aligning their maintenance approaches to the volatility patterns in their legacy software. (149 words)

**Index Terms**: Knowledge management, software maintenance management, information systems quality, software volatility, corrective software maintenance.

# Introduction

Software maintenance, the correction of errors in a system and/or enhancements to the system, is a critical process for most organizations as it is essential for the continual availability, usage and improvement of systems that allow organizations to operate [25] [7, 31]. Software maintenance requires a significant proportion of organizational resources in terms of costs and human capital [7]. Software maintenance expenditures can account for over 50% of the IT budget [7]; two-thirds of the total life-cycle cost for an application [7, 24], and over 50% of software development efforts [24, 35].

Corrective software maintenance, the portion of software maintenance involving the repairing of defects in a system, is an important part of these efforts [42]. Software maintenance is an appropriate subject for managerial attention because in addition to its critical role in organizational performance, it is well documented that software maintenance can be difficult to perform and manage [7, 31, 49]. Software maintenance requires extensive information processing by developers to understand the problem, the solution and how to implement the solution within the context of the software [5, 9, 21, 25]. Further, software maintenance is primarily driven by the knowledge of the software maintainers. Maintainers must first understand where such corrective actions need to be made [36], and then further develop the solutions to resolve these discovered errors [5, 7]. Additionally, software maintenance tends to remain a complex and difficult task as it is the least studied process in the systems lifecycle, and therefore improvement comes at a relatively slower rate than in other areas of software engineering [55].

Software maintenance is heavily reliant upon the knowledge of the maintainers and their ability to leverage their knowledge and correct software defects as they are detected [4, 5, 7]. Given the importance of knowledge in this process, we leverage research literature from *knowledge management*, which provides a general framework that can be used to theorize the more effective use of knowledge in software maintenance. This literature proposes three main approaches to knowledge-based management: technology-based, experience-based and skill-based, which we apply to corrective software maintenance [22, 32, 42].

A *technology-based* approach to knowledge sharing in software maintenance relies upon the use of tools and systems to codify knowledge within repositories or similar tools (e.g., [20]). This approach is beneficial in that knowledge is always available for maintainers and is not susceptible to direct knowledge loss when team members leave the maintenance team. Additionally, a broad range of information can readily be located within these repositories by linking with the created source code documentation, dictionaries, design diagrams, and other development documents. However, these complex technologies can require many resources to develop and update, and it can be difficult to use, store and locate all forms of desired information [20].

An *experience-based* approach to knowledge sharing in software maintenance relies upon the interaction of team members to convey knowledge about the software to other team members, based on their shared experience developed over time (e.g., [16, 20]). This approach is advantageous in that individuals may specialize within the maintenance team, and can quickly obtain information from other team members who have the desired knowledge [18, 39]. However, this approach becomes heavily dependent upon knowledge collocated within team members, and, if knowledge is uniquely held by one maintainer, the team may suffer expensive knowledge gaps if that individual leaves the team or becomes otherwise unavailable.

A *skill-based* approach to knowledge sharing in software maintenance relies upon the skills and abilities of the software maintainers to apply their expertise to relevant maintenance problems and solutions [5, 20]. This is a general approach that builds upon an understanding of expertise in software development in that more skilled maintainers are better equipped to produce higher quality software [22]. This approach relies upon the selection, attraction and continuance of highly skilled or expert software maintainers.

Previous research and practical application of these strategies have shown that each of the three approaches to knowledge sharing in corrective software maintenance can improve the ability of the team to produce higher quality software [5, 20, 22]. Each approach is used in industry and has become part of general management principles for managing corrective software maintenance [5]. However, there remains a general open question as to what degree these three approaches to managing knowledge are equally effective for different types of software maintenance tasks.

The inherent difficulty of software maintenance is compounded if the software being maintained is changing, i.e. volatile. *Software volatility* is the length of time between software modifications and the magnitude of these modifications [9]. We posit that software volatility may alter the ability of software maintainers to identify problematic parts of the code and to apply the necessary knowledge to solve given problems, and therefore can affect the efficacy of the different approaches to knowledge sharing. Software volatility can be experienced along three dimensions: frequency, predictability and magnitude [9, 10, 11, 12, 13].

*Frequency* refers to the rate or timing of maintenance tasks [12]. Some software applications are modified quite frequently with the time between maintenance tasks being determined in days, while other applications may have years between modifications. Frequency increases the overall complexity of software maintenance due to the changed timing of corrective maintenance tasks. For example, if an application is frequently modified, maintainers have to be consistently scheduled, and the ability of individual maintainers to understand all of the changes may be taxed due to the sheer number of such changes. However, infrequent changes can also increase the difficulty of software maintenance in that the maintenance team will be less familiar with the application, and will likely experience steeper learning curves when attempting to maintain the software. This can make it more difficult to reliably estimate the cost and duration of software maintenance efforts.

*Predictability* refers to the ability of managers and maintainers to estimate when maintenance tasks will need to be performed [12]. Some applications are regularly maintained (e.g., updates to an accounting application are regularly updated with new tax laws), while others are updated at unexpected intervals due to novel changes in the organization’s operating environment. As the ability to anticipate such maintenance is decreased, it becomes more complex to manage the maintenance process as preferred, or even required, resources may be unavailable when critical maintenance tasks need to be performed.

*Magnitude* refers to the overall size of the maintenance task to be performed as measured by lines of code added during the maintenance task [12]. Some modifications are extensive, while others are minor or incremental. Given that larger modification sizes tend to incorporate more code and therefore are more likely to affect interoperability, larger maintenance tasks can be expected, all else being equal, to decrease the ability of maintainers to correctly update the code, as well as the ability of management to successfully oversee such tasks.

Software volatility could change the inherent complexity associated with updating a software application, and this may have important implications for the effectiveness of the knowledge-sharing approaches adopted by management to facilitate corrective software maintenance [9, 10, 11, 12, 13]. Thus software volatility may serve as a key moderator of the relationship between technology-, experience- or skill-based approaches to knowledge sharing and corrective software maintenance performance. In this paper, we formally investigate whether this is the case, asking the following overall research question:

How do software volatility patterns moderate the effectiveness of technology-, experience- and skill-based approaches to knowledge sharing in corrective software maintenance?

Our study first demonstrates a method for managers to categorize software on several definable, objective measures of software volatility. Second, it shows how software volatility impacts the effectiveness of technology-, experience- and skill-based approaches for corrective software maintenance. Finally, recommendations for aligning knowledge sharing approaches to corrective maintenance based on the application’s pattern of software volatility are provided.

The remainder of this paper is structured as follows. The next section briefly reviews the relevant literature on cognition and software maintenance, software volatility and the general knowledge sharing approaches. We then draw upon contingency theory to pose several hypotheses on the moderating effects of software volatility patterns. Next, our research setting, data collection and measures are described prior to the analysis of the data. Finally, we discuss our results, including their implications for research and practice.

# Relevant Literature

## Cognition and Software Maintenance

*Software maintenance* has been formally defined as the activities that keep software operational and responsive after it has been implemented and placed into production, along with continued performance improvements [7]. Maintenance has traditionally been divided into four different categories: corrective, adaptive, perfective and preventive [46]. *Corrective* maintenance is a response to correct some detected failure or fault in the software, *adaptive* maintenance stems from changes in the operational environment, *perfective* maintenance is initiated by changes to the business or user requirements, and *preventive* maintenance is oriented toward future maintainability of the software. Our focus in this study is on *corrective* software maintenance, and all further references to software maintenance are limited to corrective software maintenance.

Various types of knowledge are required for the successful completion of corrective maintenance tasks [20, 23, 25]. First, before software can be maintained, maintainers must acquire knowledge about its code and structure [5, 7, 19, 21, 25]. In order to debug, correct, or improve existing software the maintainer must also typically understand the application domain and the relationships among the software, its environment, its users and the organization [19, 21]. All of this understanding typically requires significant amounts of time, effort and thinking on the part of the maintainer. As the desired knowledge necessary for comprehension of the software is distributed among the source code, documentation, and other maintainers, the search for this knowledge can be time-consuming and difficult [36]. Research on software maintenance has often highlighted that a majority of the maintainers’ effort in maintenance is spent on acquiring this understanding, with estimates suggesting over 60% of total effort being devoted to the comprehension task [1]. Additionally, studies have also found that this type of knowledge can degrade in human memory [4], which is why maintainers often rely on external memory aids [36].

Second, maintainers must also have knowledge regarding how to correct the identified error [5]. This knowledge may range from relatively basic technical knowledge to extensive knowledge required to correct a complex process that is integrated throughout an entire application [14]. Finally, maintainers must also have knowledge of where in the application the corrected code needs to be placed in order to complete the maintenance task. However, research has noted that the complexity of code increases with time, in an entropic manner, which tends to make the maintenance task more complex as a function of the age and modification history of the application [25].

Therefore, based on the prior research, corrective software maintenance can be described as an intense, cognitive, human information-processing task with various knowledge bases serving as inputs for the maintainer to utilize in identifying the faulty code and creating modified code as an output [7].

## Software Volatility – A Multidimensional View

Software applications are not equal; each one is maintained differently due to the various constraints placed on it. The differences in maintenance modifications also hinge on other factors, such as the timing of such modifications, and how predictable these changes are, and their relative sizes [11]. Such software volatility is an important dimension of the software maintenance environment and has often been cited as a common cause of software errors, which have been shown to be a driving factor of software maintenance cost and effort [15].

We adopt a multi-dimensional view of software volatility that has been proposed in previous research [9]. Specifically, as previously discussed, software volatility refers to the *frequency,* *predictability* and *magnitude* of changes in a particular piece of software [9]. Prior research has highlighted that software maintenance is not performed in isolation, but instead that each application experiences differing levels and types of volatility that may increase or decrease the level of complexity inherent in already complex maintenance tasks [9, 10, 11, 12, 13]. Further the frequency, predictability and magnitude of software maintenance are not isolated dimensions that occur without consideration for the other dimensions. Rather, software volatility consists of the joint effects produced by the combined dimensions, which may produce additive, suppressive or interactive effects. Therefore, this research uses the composite pattern of overall volatility, rather than individual dimensions in isolation [9, 10, 11, 12, 13].

## Knowledge Sharing

As previously discussed, maintainers often seek specific knowledge to complete software maintenance tasks [21]. As much of this knowledge is poorly documented, lacks comprehensible structure, or is locked in another individual’s head [21], this knowledge must be gathered from a variety of sources [28]. Likewise, prior research has characterized software maintenance as a knowledge management issue, and has applied lessons from the knowledge management literature to software maintenance [5, 20, 45]. As knowledge acquisition is difficult, there are several different approaches that can be adopted to aid maintainers in their drive to acquire necessary knowledge. Three general approaches to knowledge sharing have been addressed in the literature: *technology-based, experience-based* and *expert-based*. Each of these is discussed in turn.

### Technology-based Approaches

The technology-based approach, along with its advantages and disadvantages, is discussed in the knowledge management research literature regarding repositories and knowledge management systems. *Knowledge management systems* refer to technologies developed to support the creation, storage, retrieval, and application of knowledge [3, 27, 50]. A knowledge repository allows team members to codify their knowledge and experience into the software tool, which would allow other team members to search through content, and locate desired information at later times [3]. These types of technologies are readily available and their use by software development teams may help knowledge sharing in several key ways [32].

First, technology-based repositories allow knowledge to be captured and codified by team members and easily passed to other team members. Numerous technologies exist that aid in codifying knowledge (e.g., [20]). Second, as knowledge is then stored within the technology, the rotation of team members and their respective knowledge is no longer subject to idiosyncratic loss as relevant knowledge is retained within the technology. Thus, uniquely held information by individuals can more readily be captured and utilized by future maintainers, regardless of time or staffing changes. Last, as knowledge can more readily be reused within these technologies, the sharing of knowledge necessary for software maintenance can be readily improved, assuming that codified knowledge is clearly described and updated within the system [3, 27, 36]. By improving the sharing of knowledge to maintainers, the time necessary to understand the software should be reduced, thereby increasing the rate and quality with which maintenance tasks can be completed.

### Experience-based Approaches

The second approach that maintainers can use to obtain knowledge about the software is from members of the maintenance team itself. Knowledge to maintain software is embedded within the heads of maintainers who have created the code, commented it, and have maintenance experience with it [53]. Experienced maintainers on a team can serve as mentors for newer maintainers who join the team so that the newer maintainers can get up to speed more quickly [20]. However, new maintainers are often assigned to a team after software has been designed and implemented and after the original maintainers have moved to other projects or other organizations, and have left behind inadequate software documentation [53].

Due to the weakness of relying upon key individuals it is important for maintainers to continually share experiences with other maintainers and to ensure that this knowledge about the software is shared within the maintenance team. This knowledge sharing is commonly practiced in maintenance teams through the day-to-day exchanges and interactions between team members, and through more formal meetings, post mortem reviews, etc. [5, 20]. Experienced software maintenance teams can employ such methods to increase the sharing of knowledge among team members.

The experience-based approach to knowledge sharing within software maintenance teams, and its advantages and disadvantages, can be readily explained by the social-psychology literature regarding transactive memory systems [37, 39, 51]. *Transactive memory* refers to the notion that individuals in groups can serve as repositories of information that may be needed by the group at some point in time [51]. Specifically, transactive memory is defined as the combination of knowledge held by group members and an awareness of the knowledge held by other members in the group that is developed through shared common experiences [39].

Several studies have found that groups that are able to build transactive memory systems enjoy increased team performance, and have proposed three explanations as to why transactive memory systems are able to improve team outcomes [37, 38, 40]. First, transactive memory systems allow groups to divide knowledge storage and retrieval functions about various domains to specific individuals and to alleviate the need for each group member to encode and to be able to utilize each piece of knowledge that may be required by the group. Further, as certain individuals become responsible for specific domains of knowledge, expertise will be developed within the group that will increase the ability of the group in confronting and solving problems and tasks. Second, by removing the cognitive burden from each individual to learn all group-related knowledge, each individual has an increased capacity to devote more cognitive resources to other team relevant tasks (i.e., problem identification, solutions, etc.). As such, group members can spend less time searching for information during the task process and the group can rely upon the expert of that domain to supply the required information. Third, transactive memory systems improve the ability of group members to coordinate implicitly. As the shared understanding of who knows what increases, each team member is more able to anticipate the behaviors of other group members. This ability to anticipate the actions of other group members facilitates coordination and efficient interactions [37].

### Skill-based Approaches

The third general approach that maintainers can use to obtain knowledge regarding the software is from their own respective skill or expertise with the application itself. This approach differs from that of an experience-based approach in that each maintainer is expected to have high levels of expertise and would be able to acquire the requisite knowledge *without* the assistance of others within the team. This approach relies upon general expertise with the application, with the assumption that experts are the most equipped to deal with complexity and to engage in problem-solving mechanisms that would enable them to maintain software in complex environments [19, 22]. Previous research in software maintenance has already posited and found that developers are able to acquire such skills through their maintenance of the software; developers learn by maintaining [17]. However, this learning is not easily applied to other applications, and thus limits the skill of the developer to the application that he or she has worked on previously.

Skilled software maintainers have the previous experience and underlying mental models that allow them to more easily recognize, understand, and solve complex maintenance issues. However, the skill-based approach to corrective software maintenance can potentially suffer from several weaknesses. First, the acquisition and continuance of such skill is difficult and costly for an organization. Highly skilled maintainers of a given application are not easy to come by via hiring, and therefore require considerable time and experience with the application itself to develop their expertise [17]. Developing expertise with a software application is a costly investment, which makes the organization highly dependent upon that maintainer. This reliance upon expertise increases the risk to the organization in the event that such an expert becomes unavailable (e.g., due to illness, withdrawal from the organization, or temporary assignment).

In addition to the above weaknesses, a limitation of relying upon experts in a particular software application is the potential for missed opportunities. Skilled maintainers are more likely to rely upon heuristic or habitual reasoning and may overlook potentially disruptive, radical or creative approaches to maintenance [17].

# Hypotheses Development

We draw upon contingency theory to pose hypotheses about the moderating effects of software volatility on the effectiveness of the different knowledge sharing approaches in corrective software maintenance. A multidimensional view of software volatility has a parallel with a similar view of environmental variability from organizational theory [52]. According to contingency theory, different external conditions might favor different organizational design approaches, such that the effectiveness of the approaches is moderated by environmental variables [47]. Although contingency theory was initially developed at the organizational level, it has been shown to also apply to the project level as projects can also be seen as “temporary organizations within organizations” [56].

Taking a contingency approach to software volatility as an external variable at the project level (e.g., a software application), we posit that the effectiveness of the technology-based, experience-based and skill-based approaches to corrective software maintenance will vary according to the pattern of volatility for the application. Essentially, each knowledge sharing approach has distinct advantages and disadvantages, which may become more salient given the software application’s volatility pattern, thereby altering the effectiveness of the approach.

As software can experience its own pattern of volatility, it is important to understand volatility and its potential effects on software maintenance tasks, and the knowledge required to complete such tasks correctly. A logical question that arises is whether one of the approaches to knowledge sharing is more effective than the other for a particular application, given its level of software volatility? In the following sections we posit that software volatility differentially influences the relationship between the technology-based, experience-based and skill-based approaches to knowledge sharing and corrective software maintenance performance. Thus, it is not which approach is best, but which approach is *best suited* for the software application, given the pattern of software volatility that it experiences.

We now develop our hypotheses regarding the effects that will most likely be exerted by software volatility on the relationship between knowledge sharing approaches and corrective maintenance performance. We conceptualize software maintenance performance in terms of software quality (i.e., software errors) as software quality is widely accepted as an indicator of software maintenance performance and is particularly relevant to corrective maintenance [41, 44, 55].

Figure 1 illustrates our research model. The predicted impact of software volatility on the effectiveness of the different corrective maintenance strategies is shown given the known frequency, predictability and magnitude of modifications to the applications.

<<INSERT FIGURE 1>>

## Frequency of Software Maintenance and Effectiveness of Knowledge-Sharing Approaches

Knowledge can more easily be recalled and utilized from maintainers’ memories if modifications occur more frequently [3]. Memory recall for shorter periods of time is much more efficient and precise for smaller intervals than longer ones [43, 55]. Even though maintainers could also query technology-based repositories to acquire this information, it is faster for the maintainer to simply recall this knowledge, as it has been accessed more recently, and is thus easier to recall [3]. The amount of effort required to articulate the knowledge and search for it within the technology can be expected to take longer than recalling the more accessible knowledge from the maintainer’s own memory.

We thus expect that the relative advantage of memory recall for frequent modifications should hold for both experienced teams, as team members could take advantage of their transactive memory systems with little or no memory decay from the last period of usage, and for skilled maintainers as memory decay is likewise expected to be minimal [39]. This suggests that either “person-based” strategy (experience- or skill-based) will be more effective when the frequency of maintenance is higher. Corrective maintenance teams that experience little turnover, or have spent considerable amounts of time together, are more likely to be aware of other maintainers’ expertise and to rely upon them to make requisite knowledge available as needed. With frequent maintenance team members can acquire the knowledge that is needed more easily as it is more readily available and accessible from other team members.

Likewise, skilled maintainers will have readier access to information in memories, experience less memory decay and will process more potential cues for memory recall, making more frequent modifications more favorable for skilled maintainers. Previous work in corrective software maintenance has found that teams are able to successfully train, mentor and bring up to speed other maintainers to increase the success of corrective maintenance efforts [20]. Likewise, work has also shown that skilled maintainers are more able to recall information when it was recently stored or acquired [38].

Thus, when corrective software maintenance is more frequent, we expect that the effectiveness of both experience- and skill-based approaches to knowledge sharing will be enhanced more than will technology-based approaches. As described above, with frequent maintenance, the knowledge in human memory is more recent and less prone to memory degradation and decay, and can be more easily accessed, all else being equal. This should help skilled individuals and experienced teams to be more accurate in pinpointing the location in the application where the correction needs to occur and in making the actual change required to fix the problem without introducing new errors. Therefore, we posit that:

H1a. When corrective software maintenance is more frequent, the effectiveness of skill-based approaches to knowledge sharing will be enhanced more than that of technology-based approaches in terms of reducing software errors.

H1b. When corrective software maintenance is more frequent, the effectiveness of experience-based approaches to knowledge sharing will be enhanced more than that of technology-based approaches in terms of reducing software errors.

In contrast, when corrective maintenance is infrequent, technology-based approaches may be more effective than person-based approaches. Knowledge can be more accurately stored and recalled over longer periods of times through technology-based repositories rather than relying upon human memory [3]. Human memory can be error-prone and information can degrade, be inaccessible for periods of time, or simply be incorrectly recalled, all of which would reduce the performance of maintenance tasks. In order for these tasks to succeed the maintainer must acquire, maintain and use all requisite pieces of knowledge.

It is therefore important that acquired knowledge be maintained until the maintainer requires it to write and/or test code for the maintenance task. Repositories do not exhibit problems from human memory decay or recall failures, and, with advanced search engines, maintainers should be able to locate desired information with increased ease [20]. In other words it becomes more difficult to rely upon human memory if the software is maintained sporadically, as it is less likely to be correctly recalled [29]. The recall of such memory is more likely to succeed through the use of technical storage solutions that do not decay over time [20].

Previous studies have found that knowledge needed for maintenance quickly degrades in the maintainer’s memory [4], which is why maintainers rely on external memory aids like a knowledge repository, and maintainers often refer to technology-based aids when writing code for maintenance tasks [36]. By relying upon repository tools it is more likely that maintainers will be able to locate required information and successfully complete maintenance tasks [20]. The use of such tools enables the software maintainer to locate correct knowledge as needed, rather than relying upon their own memory, which is susceptible to memory recall failures and biases. The usage of the repository for infrequent modifications enables the maintainer to use correct information, and thereby successfully complete the maintenance task. Therefore, for the reasons above, we would expect that technology-based approaches would afford additional benefits for applications with a low frequency of modification due to the ability to find accurate information and avoid incorrectly recalled information from memory:

H2a. When corrective software maintenance is less frequent, the effectiveness of technology-based approaches to knowledge sharing will be enhanced more than that of skill-based approaches in terms of reducing software errors.

H2b. When corrective software maintenance is less frequent, the effectiveness of technology-based approaches to knowledge sharing will be enhanced more than that of experience-based approaches in terms of reducing software errors.

### **Predictability of Software Maintenance and Effectiveness of Knowledge Sharing Approaches**

When applications are modified more frequently, person-based approaches to corrective maintenance (both experience-based and skill-based approaches) are proposed to be more effective. This section considers how the predictability of the corrective maintenance task may impact the effectiveness of the different person-based approaches based upon the availability of knowledge resources.

The experience-based approach to knowledge sharing has a significant disadvantage for unpredictable modification patterns when compared to the skill-based approach in that knowledge is not based in any one team-member, but is a collection of all knowledge held by the team [20, 39]. By spreading out knowledge to various members of the team it is expected that whenever a corrective maintenance issue is identified, the requisite knowledge will be held somewhere within the group memory and will thus be available for recall. However, with unpredictable maintenance tasks, it is possible that one or more members of the team may be unavailable. This would hamper maintenance efforts if unavailable team members held unique knowledge that is now inaccessible to the team [39, 40, 41].

The threat of unavailable knowledge resources is less likely when relying upon the general, independent skill or expertise of individual maintainers, rather than cohesive transactive teams because transactive memory systems have an increased tendency to encourage members of a team to specialize, more so than would be found if management had relied upon a skill-based approach [38, 39, 40]. As team members train together and ascertain expertise distribution within the members of the team, group members’ transactive memory systems encourage specialization of knowledge within team members. Whereas for the skill-based approach required skill sets for maintenance are more likely to be redundantly found within the team as each maintainer is striving to improve his or her skill set without regard to what others in the team know [38, 39].

Researchers have long emphasized that software maintenance is highly unpredictable [25], and that software maintenance teams serve as efficient means to transfer necessary information for maintenance projects, and to provide an effective coordination mechanism for the team and thereby provide effective maintenance, assuming that the essential team members are available [20]. Given the risk associated with the availability of requisite members in the experience-based approach, and the relative ease to assign highly skilled maintainers to a corrective maintenance task whenever it is needed, we propose that:

H3. When corrective software maintenance is more unpredictable, the effectiveness of skill-based approaches to knowledge sharing will be enhanced more than that of the experience-based approach, in terms of reducing software errors.

### **Magnitude of Software Maintenance and Effectiveness of Knowledge-Sharing Approaches**

Larger modifications tend to be more complex and thus require more knowledge for maintainers to understand the software and the desired changes. As the size, history and length of the software increases, so does its complexity [25, 55]. A large software modification requires more knowledge to comprehend it, which is necessary before any maintenance on the software can be performed. Further, every time software is modified, it grows in complexity [25]. With the passage of time and increased maintenance efforts of maintainers the software tends to become more and more complex if nothing is done to control the complexity [7]. In addition, larger modifications are assumed to be inherently more complicated as they require more time and effort to understand the code and the inter-relationships between elements in the code [25, 55]. Thus, the complexity can be reduced if the maintainer can more readily and easily acquire knowledge about the software and its relationships with other entities.

Given that each of the knowledge sharing approaches reduces the overall complexity of large modifications by increasing the ability of the maintenance team to have the requisite knowledge required for corrective software maintenance, we expect that the effectiveness of all approaches will be enhanced for larger modifications compared with smaller modifications due to the ability of each approach to reduce the complexity of these larger modifications.

H4: When corrective software maintenance is of a larger magnitude, the effectiveness of technology-based (H4a) experience-based (H4b) and skill-based (H4c) approaches to knowledge sharing will be enhanced in terms of reducing software errors when compared to maintenance of a smaller magnitude.

# Research Model and Methodology

## Research Setting and Data Collection

Data were collected from a large company in the Midwestern United States. The company has a centralized IS department that serves as an efficient and effective point of access for data collection. The IS department has separate development and maintenance units, which contribute to its ability to better control, manage, and measure its maintenance activities. Given the extensive longitudinal data on software maintenance tracked by the company, the company has been the subject of prior studies, resulting in a set of research findings on different aspects of software evolution [9, 10, 11, 12, 13]. That prior work has laid the foundation for this final study in the research program.

Many of the maintenance unit employees have worked with the unit for a significant amount of time (9 years on average), and many of the unit’s supervisors have been with the unit for several decades. Due to the long tenures of both the employees and supervisors within the maintenance unit the collected dataset is of consistently high quality and extends over a long period of time. We extracted software maintenance information from histories and logs that were written by the maintainers each time a modification was performed. Logs were kept for more than 25,000 changes to 3,800 modules in 21 different business applications [35]. Additionally, information was also extracted from the company’s code versioning system.

The company does not use a formal release cycle for changes to each application. However, when errors occurred during nightly batch runs, corrective changes were implemented as soon as possible to fix discovered errors and avoid repetition of such errors. Changes were typically implemented on a monthly basis, and all of the error reporting at the firm was done on a monthly basis. Given the nature of the data, our longitudinal time interval is thus one month. Our dataset in this study encompasses a three-year range with data reported at the monthly level for 21 applications, for a maximum possible sample size of 756 observations. However, given that we are analyzing the data longitudinally, following the standard practice, we have lagged time-varying variables by one-month[[1]](#footnote-1), resulting in an overall sample size of 600 [26].

## Dependent Variable

We use the number of errors to measure the level of quality for software maintenance, in accordance with previous literature [1, 13, 46, 48]. Software error rates (ERROR) were measured on a monthly basis and consist of the operational errors that occurred during nightly batch procedures run for each application that resulted in system failures during execution (*i.e.*, errors do not refer to minor faults or cosmetic issues). The operational errors from the nightly batch execution of each application were summed for each month. This monthly sum was also log transformed to correct for the skewness of this variable for each application [26].

## Independent Variables

### Software Volatility

As previously defined, software volatility consists of three distinct dimensions: frequency, predictability and magnitude, and we build on previous research for the measurement of these dimensions as described in [11]. Frequency was measured as the time between software modifications, as measured in days[[2]](#footnote-2). Predictability was measured as the variance in the time between modifications (i.e., frequency). This measure was also adjusted to account for the increased variance that occurs in older applications; we thus divide by the square of the application’s age, to minimize this effect. Magnitude was measured as the total size of the software modification divided by the total size of the system.

In order to discern the potential patterns that these three dimensions may form, we dichotomize each dimension as high or low for each application-month in our dataset. The volatility pattern was categorized as low if its score fell below the mean for the measure of volatility in comparison to all other applications at that time period. For example, for application 1, during the first month in the dataset its modification frequency was above the average (standardized frequency of .489, which resulted in a score of 1 for frequency); the variability of its modifications was more irregular than for other applications (standardized predictability of -.268, which resulted in a score of 0 for predictability); and its modifications tended to be smaller than other applications at that time (standardized magnitude of -.207, which resulted in a score of 0 for magnitude).

By dichotomizing the three dimensions for each application month, a total of eight patterns (i.e., frequency x predictability x modification size or 23) would be possible. However, our dataset did not exhibit all possible combinations of the three software volatility variables. Rather we only found four such software volatility patterns in our dataset. Table 1 shows the mapping of the three dimensions to the software volatility patterns. To represent the patterns shown in Table 1, we created three binary variables to reflect the assignment of a software application to a particular volatility pattern in a particular time period. The variable for the second volatility pattern (P2) is set to “1” if the application has low frequency, high predictability and small magnitude of modification, otherwise it is “0”. The variable for the third volatility pattern (P3) is set to “1” if the application has low frequency, high predictability and large magnitude of modification, otherwise it is “0”. The variable for the fourth volatility pattern (P4) is set to “1” if the application has high frequency, high predictability and large magnitude of modification, otherwise it is “0”. Finally, the first volatility pattern (P1) is maintained as the base case, when a “0” is entered for the P2, P3, and P4 variables.

<< INSERT TABLE 1 HERE >>

### Technology-based Approach

The technology-based approach relies upon the use of tools to store information and allow maintainers to quickly locate desired information. The tool used by this company is CA-TELON[[3]](#footnote-3); a full-service, complete life cycle development tool that provides a design facility, code generation and code repository.

The tool used by the company creates source code based upon design information entered into the tool. For example, a maintainer could design a layout for a screen or report using the tool and the tool would then automatically generate the software code to implement the screen or report. The tool thus captures designs as well as the related source code in its repository, and both designs and code could be searched and modified or re-used. Since, in addition to a code generator, the tool has both a design facility as well as a repository for code; it is used to store important information about the application that is useful to maintainers. Maintainers learn about the application and access the code using the tool. Thus, the tool affords a technology-based approach to knowledge, and the usage of this repository tool is therefore a measure of the technology-based approach [30, 33].

The technology usage variable (TECH) was created by using the proportion of an application’s code that was created or maintained using the repository tool versus the total amount of the application code. This measure is a good proxy of tool use. For applications where code has been created using the tool, the maintainers must use the tool to view and maintain the code. Therefore, we would expect that two applications having a similar use of tool (e.g., similar proportion of code created by the tool), would require a similar reliance upon the tool for knowledge sharing, and vice versa. This variable was normalized for each application and lagged one month. As noted earlier, the time-varying independent variables, such as this one, are lagged by one time period to mitigate the effects of potential endogeneity issues [26]. This variable was also interacted with the software volatility patterns to account for any interactive properties that this strategy may have with a given volatility dimension as we have predicted in our hypotheses.

### Experience-based Approach

As previously discussed, the theoretical explanation for the experience-based approach relies upon transactive memory systems. In accordance with previous research [38, 39], we use the familiarity of the team members as a surrogate measure for transactive memory systems. The team member experience index (EXP) was created by averaging the total number of days that team members had been maintaining the application as a team. In other words, EXP is an average of the number of days that all members of the team have worked together with each team member on the application prior to the current month. This is a standard approach for the measurement of familiarity for transactive memory systems in social psychology [39]. This variable was then normalized[[4]](#footnote-4) for each application and lagged one month. It was also interacted with the software volatility patterns to account for any interactive properties that this approach may have with a given volatility dimension, as we have predicted in our hypotheses.

### Skill-based Approach

We define *maintainers’ skill* (SKILL) to represent the level of coding expertise as rated by the team supervisor for each team member (on a scale of 1 – Low to 5 – High), averaged for the team. This variable was also normalized for each application and lagged one month. Additionally, since our final model is focused on showing how the maintenance approach is altered by the patterns of software maintenance, SKILL is also interacted with each of the software volatility patterns described earlier.

## Control Variables

While the focus of our research is on the interaction between management approaches and volatility, we also collected a set of widely agreed upon *control variables* which could be expected to have an impact on software maintenance quality. Each is briefly described below.

### Application Age

Applications that are older tend to be more complex, all else being equal, given the increased amount of modifications and alterations that they have experienced [7]. Given that the increased complexity of the application may increase the likelihood for errors, age is used as a control variable [6, 34]. Application age (AGE) is measured by the number of months since its original release. This variable was normalized for each application, and lagged one month.

### Application Size

Application size can also alter the quality of software maintenance and the relative density of the error rate. We used two measures as surrogates for the size of the application. First, we determined the function points (APPFP) for the application at the end of each month [8, 54]. This variable was then normalized for each application, and lagged one month. Second, we controlled for the lines of code (LOC) that an application had at the end of each month [8, 43]. This variable was then normalized for each application, and lagged one month.

### Application Complexity

We controlled for the complexity of the application, which was determined through the use of a commercial code analysis tool. The complexity metric (COMPL) was calculated for each application by counting the total number of data elements referenced in the application and dividing by the number of modules in the application at the end of each month [9]. This variable was also normalized for each application, and then lagged one month.

### Application Usage

Applications that are more heavily used are more likely to be modified more frequently and have more modification requests. We use the number of online transactions during time period *t* (TXNUMt) as a measure for application usage. The measure assesses the number of transactions of the application initiated by the users online. This variable was also normalized for each application. The descriptive statistics for these variables are summarized in Table 2.

<< INSERT TABLE 2 HERE >>

## Empirical Model

Although the hypotheses only predict how each software volatility dimension affects the efficacy of each approach in regards to a single dimension, we provide a complete model to show how including all of the dimensions of software volatility can be expected to alter the effectiveness of a knowledge sharing approach in increasing maintenance quality. As we have described earlier and have shown in Table 1 [11], applications may be assigned to one of four different software volatility patterns (P1-P4). We analyze our model based on these volatility patterns in Equation (1) below[[5]](#footnote-5):

ERRORt = β0 + β1P2i + β2P3i + β3P4i + β4TECHi(t-1) + β5EXPi(t-1) + β6SKILLi(t-1) + β7TECHi(t-1)xP2i + β8TECHi(t-1)xP3i + β9TECHi(t-1)xP4i + β10EXPi(t-1)xP2i + β11EXPi(t-1)xP3i + β12EXPi(t-1)xP4i + β13SKILLi(t-1)xP2i + β14SKILLi(t-1)xP3i + β15SKILLi(t-1)xP4i + β16AGEi(t-1) + β17APPFPi(t-1) + β18LOCi(t-1) + β19COMPLi(t-1) + β20TXNUMit + εit (1)

# Analysis and Results

Our data form a cross-sectional time series panel, dimensioned by application *i* and month *t*. Accordingly, we used a time series generalized least squares method with heteroskedastic, but uncorrelated, error structure across panels and correction for autocorrelation at the monthly level for all analyses [26]. We now report the results of our model and evaluate our hypotheses.

## Software Volatility Model Analysis

Table 3 shows the results for our model (see Equation 1) and sample (n=600). The Wald’s Chi-squared fit statistic indicates that our model is a good fit to the data and is also highly predictive of the error rates (p = .000).

<< INSERT TABLE 3 HERE >>

## Hypotheses Tests

Our first set of hypotheses (H1a and H1b) pose that the effectiveness of person-based approaches to knowledge sharing (i.e., skill- and experience-based approaches) will be enhanced more than that of technology-based approaches when corrective software maintenance is more frequent. To evaluate H1a and H1b, we compare the coefficients for skill (SKILL) (H1a) and experience (EXP) (H1b) versus technology (TECH) for application volatility pattern P1, which includes applications with a high frequency of modification[[6]](#footnote-6). This comparison reveals support for H1a as the skill-based approach (β6 - β4 = -.1820, t = 10.95; p < .001) is more effective in terms of lower software errors than the technology-based approach for high frequency modification patterns. However, this is not the case for the experience-based approach as (β5 - β4 = .0825, t = 5.35, p < .001) and H1b is not supported.

Our second set of hypotheses (H2a and H2b) posit that the effectiveness of the technology-based approach to knowledge sharing will be enhanced more than that of person-based approaches (i.e., skill- and experience-based approaches) when corrective software maintenance is less frequent. To evaluate H2a and H2b, we compare the coefficients on the interactions of TECH, EXP and SKILL with P2; and the interactions of TECH, EXP and SKILL with P3, since application volatility patterns P2 and P3 include applications with a low frequency of modification. This comparison reveals full support for both H2a and H2b as technology-based approaches are more effective than skill- and experience-based approaches for applications with a low frequency of modification (for P2 (H2a) β7 – β13 = -.9765, t = 40.08, p < .001, and P2 (H2b): β7 – β10 = -.5128, t = 26.46, p < .001; for P3 (H2a) β8 – β14 = -1.3877, t = 62.00, p < .001 and P3 (H2b): β8 – β11 = -1.4845, t = 69.00, p < .001 ).

Our third hypothesis (H3) posits that the effectiveness of skill-based approaches to knowledge sharing will be enhanced more than the experience-based approach when corrective software maintenance is more unpredictable. To evaluate H3, we compare the coefficients on the interactions of EXP and SKILL with P1 since this application volatility pattern includes applications with a low predictability of modification. This comparison reveals full support for H3 as the error rate for the experience-based approach is significantly less than that of the skill-based approach for unpredictable modification patterns (β5 – β6 = -.2645, t = 15.78; p < .001).

Our final set of hypotheses (H4a-c) posit that when corrective software maintenance is of a larger magnitude, the effectiveness of H4a) technology- H4b) experience- and H4c) skill-based approaches to knowledge sharing will be enhanced in terms of reducing software errors when compared to maintenance of a smaller magnitude. To evaluate H4a-c, we compare the interactions for application volatility patterns P3 versus P2, as the only difference in the patterns is the magnitude of modification (since P3 has larger modifications than P2, software errors should be lower for the different knowledge sharing approaches for P3 versus for P2). Our analysis reveals support for hypotheses H4a and H4c as the error rates for both the technology-based (β8 – β7 = -.8569, t = 38.88; p < .001) and skill-based (β14 – β13 = -.4457, t = 18.06; p < .001) approaches are significantly lower for modifications of larger magnitude. However, H4b is not supported for the experience-based (β11 – β10 = .1148, t = 6.11; p < .001) approach. Tables 4 and 5 summarize the results of our hypothesis tests.

<< INSERT TABLES 4 & 5 HERE >>

## Graphs of Moderation Effects

In order to visualize the exact nature of the moderation between the software maintenance approach and software volatility pattern we graphically depict the moderating effects to aid the reader in understanding the relationships (See Figure 2) [26]. In order to create these graphs, we used the regression results reported in Table 3 (Full Model), and performed the following calculations. In order to calculate a “low” condition for the technology-, experience- and skill-based approaches we held all other variables constant within the model and modeled the specific variable one standard deviation below its mean. The same was performed for the “high” condition by using one standard deviation above the mean. This is the standard approach in the literature for depicting interaction effects [2]. Figures 2 and 3 graphically depict the moderation effects. We discuss our results and their graphical illustration in the next section.

<<INSERT FIGURES 2 & 3 HERE>>

# Discussion

We first discuss our main results and hypotheses tests. We then evaluate the performance implications of the indicated approach that “fits” each software volatility pattern.

## Discussion of Results from Hypothesis Tests

As shown in Table 5 the majority of our hypotheses were supported. Further, the effectiveness of the technology-, experience- and skill-based approaches to corrective software maintenance as moderated by application volatility is depicted in Figures 2 and 3. After having controlled for other factors that may have impacted errors, we found that a greater frequency of modification enhances the effectiveness of skill-based approaches to knowledge sharing in software maintenance more than the technology-based approach, in terms of reducing errors (supporting H1a)[[7]](#footnote-7).

Conversely, a lower frequency of modification enhances the effectiveness of the technology-based approach of knowledge sharing in software maintenance in terms of reducing errors compared to both skill-based (H2a) and experience-based (H2b) approaches. H2a and H2b are supported as technology-based approaches are more effective than either person-based approach in terms of reduced error-rates for infrequently maintained software.

Further, we find that the predictability of the modifications does alter the effectiveness of the person-based approaches. Specifically, we find that the effectiveness of skill-based approaches is enhanced for unpredictably modified applications (H3), which is supported.

Finally, we find that the magnitude of the modifications makes a significant impact upon the effectiveness of two of the three approaches (H4a (technology) and H4c (skill)). However, we find that the impact of the magnitude of the modification on the effectiveness of the experience-based approach (H4b) is not significant[[8]](#footnote-8).

Our results indicate that each approach’s effectiveness is, in fact, moderated by software volatility and fits with the different patterns found among the three software volatility dimensions, as explored in this paper. We do find that each software volatility pattern exhibited one approach that produced significantly lower error rates when compared to the other approaches, holding all other variables constant (See Figure 2). Specifically, our results suggest that when applications are infrequently modified (Pattern 1), managers should rely upon highly skilled maintainers to produce higher quality software. We thus recommend that managers rely upon a skill-based approach when software is frequently, yet unpredictably maintained in small units.

Our results also imply that technology-based approaches appear to produce better quality software when modifications are performed infrequently (See Patterns 2 and 3 in Figure 2). In alignment with our reasoning based on the knowledge management literature, individual software maintainers or experienced teams are not equipped to accurately recall information needed for these types of tasks, given the problems introduced by this knowledge decay. We recommend that managers rely upon the technology-based approach when maintenance is performed infrequently.

Finally, our results imply that an experience-based approach should be adopted for applications that are frequently modified at predictable points in time (See Pattern 4 in Figure 4). This fit is in accordance with our predictions. Thus, managers should rely upon the experience-based approach for frequently modified software that is predictable in its maintenance schedule.

# Conclusions

Given the complexity of the maintenance process, knowledge of the software is critical to perform the maintenance task. Knowledge of the software allows the maintainer to more quickly understand it and thus locate desired knowledge required for the modification [16, 20]. This knowledge of the software saves time and allows maintainers to be more productive.

Building on three general approaches to knowledge management, managers may adopt different approaches to store and disseminate knowledge in corrective software maintenance: technology-, experience- or skill-based. Given that each of these approaches has distinct advantages and disadvantages, this study sought to understand whether there is some differential value, or fit, between software maintenance and the general knowledge sharing approaches, based on the pattern of volatility for the application being modified. The results indicate that each of the approaches does, for a typical application at our research site, produce positive results in terms of decreased errors. However, by considering the type of software volatility that an application experiences, our results reveal that some approaches are relatively more beneficial. This allows us to provide recommendations to managers, based on the software volatility patterns, as to which type of approach should be followed for the maximum benefit.

 Our research contributes to the literature on software maintenance management in multiple ways. First, it provides a way to categorize software based on several definable, objective measures of software volatility. Managers may assess application histories and ascertain the levels of frequency, predictability and magnitude of modifications. Then, based on this assessment, the software may be compared to the established software volatility patterns to define what type of volatility the software is experiencing. By knowing what type of volatility pattern an application is experiencing, managers can utilize an approach that has a better fit with the volatility pattern and thereby increase maintenance quality.

Second, we discussed the trade-offs between the three identified approaches to knowledge sharing. The technology-based approach is effective at maintaining all codified information and allowing any maintainer to access it when needed; however this approach can require a large amount of capital and time to search and locate the relevant knowledge. In contrast, the team-based approach allows teams to quickly and fluidly acquire information from members within a team and to quickly exchange ideas. However, this approach is severely limited when maintainers do not share knowledge with team members. Uniquely held knowledge puts teams at serious disadvantages when required knowledge is unavailable due to the knowledgeable individual leaving or being assigned to another team. Lastly, we find that the practice of hiring exceptionally skilled software maintainers is very effective when maintenance on applications is frequent, yet difficult to predict in advance.

Third, we showed how factors of software volatility alter the effectiveness of each approach. Given that applications are not all the same, it is important to consider how they are maintained. If modifications occur at longer intervals of time, maintainers would be less familiar with the application and require more time and knowledge to understand the application prior to its modification. Thus, a technology-based approach would better aid these maintainers, as the technology would not suffer from memory decay. If applications are unpredictable in their need for modifications, relying upon skilled maintainers for modifications greatly increases the ability of management to increase the effectiveness for these applications by having a pool of developers that would be accessible and ready to maintain the application. Finally, for software that is has a history of large modifications, we find that technology- and skill-based approaches are better suited for these types of modifications (See Figure 3). Likewise, we note that the experience-based approach is able to produce high quality software with large modifications as hypothesized, but it also appears to be as helpful for small modification histories (See Figure 3).

Finally, we provide recommendations for approaches for maintaining software, which is especially important considering that managers may not intuitively utilize the proper approaches. Our analysis revealed that careful consideration needs to be given to the type of volatility that the software experiences to determine the most appropriate type of knowledge sharing approach.

This study has limitations common to most empirical software engineering research. First, the data were obtained from one company and therefore may or may not generalize to other companies, contexts or industries. In particular, the relatively high tenure level of employees within this company may be an important distinction between the data site and many other software maintenance organizations. However, this organization has a relatively standard set of business applications and therefore is likely to be representative of a large class of commercial users of information technology. Also, use of data from a single source has the advantage that it controls for other possible sources of variance that could affect the results that are not accounted for in the model.

Second, given the nature of commercial empirical data, maintenance approaches have been measured by the length of team rotations and the use of one tool that was utilized by the company rather than, for example, attempts to have maintainers report their actual perceptions of their transactive memory systems such as might be used in a laboratory experiment. Our measurement approaches were necessary for the longitudinal analysis as they were the only consistent measures of the development teams over the years. However, even these measures demonstrated significant results, suggesting the presence of strong effects for these knowledge sharing approaches. Additionally, it is likely impossible to determine the exact impacts of the technology-approach, as teams working together in a common goal will, by their very nature, rely upon some form of transactive memory in the team-based approach.

Future research could also determine whether the type of knowledge sharing approach and software volatility pattern alter the cost of maintaining the application. Finally, future research could further extend our findings to other organizations or industries. As business and society become increasingly dependent on software-based systems, understanding how to maintain and improve their quality over time takes on additional significance. Longitudinal studies such as this one provide a means to make positive changes in the management of these systems.

# References

1. Agrawal, M. and Chari, K. Software effort, quality, and cycle time: A study of CMM level 5 projects. *IEEE Transactions on Software Engineering* 33, 3 (2007), 145-156.

2. Aiken, L.S. and West, S.G. *Multiple Regression: Testing and Interpreting Interactions*. Newbury Park, London: Sage, 1991.

3. Alavi, M. and Leidner, D.E. Knowledge management and knowledge management systems: Conceptual foundations and research issues. *MIS Quarterly* 25, 1 (2001), 107-136.

4. Altmann, E.M. Near-term memory in programming: A simulation-based Analysis. *International Journal of Human-Computer Studies* 54, 2 (2001), 189-210.

5. Anquetil, N., de Oliveira, K.M., de Sousa, K.D., and Dias, M.G.B. Software maintenance seen as a knowledge management issue. *Information and Software Technology* 49, 5 (2007), 515-529.

6. Banker, R.D., Datar, S.M., Kemerer, C.F., and Zweig, D. Software errors and software maintenance management. *Information Technology and Management* 3, 1-2 (2002), 25-41.

7. Banker, R.D., Davis, G.B., and Slaughter, S.A. Software development practices, software complexity, and software maintenance performance: A field study. *Management Science* 44, 4 (1998), 433-450.

8. Banker, R.D., Kauffman, R.J., and Kumar, R. An empirical test of object-based output measurement metrics in a computer aided software engineering (CASE) environment. *Journal of Management Information Systems* 8, 3 (1992), 127-150.

9. Barry, E.J., Kemerer, C.F., and Slaughter, S.A. Environmental volatility, development decisions and software volatility: A longitudinal analysis. *Management Science* 52, 3 (2006), 448-464.

10. Barry, E.J., Kemerer, C.F., and Slaughter, S.A. On the uniformity of software evolution patterns. *Proceedings of ICSE*, 2003, 106-113.

11. Barry, E.J., Kemerer, C.F., and Slaughter, S.A. Toward a detailed classification scheme for software maintenance activities. *Proceedings of AMCIS*, 1999.

12. Barry, E.J. and Slaughter, S.A. Measuring software volatility: A multi-dimensional approach. *Proceedings of ICIS*, 2000, 412-413.

13. Barry, E.J., Slaughter, S.A., and Kemerer, C.F. An empirical analysis of software evolution profiles and outcomes. *Proceedings of ICIS*, 1999, 453-458.

14. Bendifallah, S., and Scacchi, W. Understanding software maintenance work. *IEEE Transactions on Software Engineering* 13, 3 (1987), 311-323.

15. Bennett, K.H. Software evolution: Past, present and future. *Information and Software Technology* 39, 11 (1996), 673-680.

16. Berlin, L.M., Jeffries, R., O'Day, V.L., Paencke, A., and Wharton, C. Where did you put it? Issues in the design and use of a group memory. *Proceedings of the SIGHCI Conference on Human Factors in Computing Systems*, 1993, 23-30.

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

18. Brandon, D.P. and Hollingshead, A.B. Transactive memory systems in organizations: Matching tasks, expertise, and people. *Organization Science* 15, 6 (2004), 633-644.

19. Chhabra, J.K., Aggarwal, K.K., and Singh, Y. Measurement of object-oriented software spatial complexity. *Information and Software Technology* 46, 10 (2004), 689-699.

20. Cubranic, D., Murphy, G.C., Singer, J., and Booth, K.S. Hipikat: A project memory for software development. *IEEE Transactions on Software Engineering* 31, 6 (2005), 446-485.

21. Dias, M.G.B., Anquetil, N., and de Oliveira, K.M. Organizing the knowledge used in software maintenance. *Journal of Universal Computer Science* 9, 7 (2003), 641-658.

22. Earl, M. Knowledge management strategies: Toward a taxonomy. *Journal of Management Information Systems* 18, 1 (2001), 215-233.

23. Faraj, S. and Sproull, L. Coordinating expertise in software development teams. *Management Science* 46, 12 (2000), 1554-1568.

24. Gibson, V.R. and Senn, J.A. System structure and software maintenance performance. *Communications of the ACM* 32, 3 (1989), 347-358.

25. Gill, G.K. and Kemerer, C.F. Cyclomatic complexity density and software maintenance productivity. *IEEE Transactions on Software Engineering* 17, 12 (1991), 1284-1288.

26. Greene, W.H. *Econometric Analysis*. Upper Saddle River, NJ: Prentice Hall, 2003.

27. Griffith, T.L., Sawyer, J.E., and Neale, M.A. Virtualness and knowledge in teams: Managing the love triangle of organizations, individuals, and information technology. *MIS Quarterly* 27, 2 (2003), 265-287.

28. Guimaraes, T. Managing application program maintenance expenditures. *Communications of the ACM* 26, 10 (1983), 739-746.

29. Holden, S.J.S., and Vanhuele, M. Know the name, forget the exposure: Brand familiarity versus memory of exposure context. *Psychology & Marketing* 16, 6 (1999), 479-496.

30. Iivari, J. Why are CASE tools not used?. *Communications of the ACM* 39, 10 (1996), 94-103.

31. Kafura, D. and Reddy, G.R. The use of software complexity metrics in software maintenance. *IEEE Transactions on Software Engineering* 13, 3 (1987), 335-343.

32. Kankanhalli, A., Tan, B.C.Y., and Wei, K.-K. Contributing knowledge to electronic knowledge repositories: An empirical investigation. *MIS Quarterly* 29, 1 (2005), 113-143.

33. Kemerer, C.F. How the Learning Curve Affects CASE Tool Adoption. *IEEE Software* 9, 3 (1992), 23-28.

34. Kemerer, C.F. Software complexity and software maintenance: A survey of empirical research. *Annals of Software Engineering* 1, 1 (1995), 1-22.

35. Kemerer, C.F. Reliability of function points measurement: A field experiment. *Communications of the ACM* 36, 2 (1993), 85-97.

36. Kemerer, C.F. and Slaughter, S.A. An empirical approach to studying software evolution. *IEEE Transactions on Software Engineering* 25, 4 (1999), 493-509.

37. Ko, A.J., Myers, B.A., Coblenz, M.J., and Aung, H.H. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *IEEE Transactions on Software Engineering* 32, 12 (2006), 971-987.

38. Lewis, K. Knowledge and performance in knowledge-worker teams: A longitudinal study of transactive memory systems. *Management Science* 50, 11 (2004), 1519-1533.

39. Liang, D.W., Moreland, R.L., and Argote, L. Group versus individual training and group performance: The mediating role of transactive memory. *Personality and Social Psychology Bulletin* 21, 4 (1995), 384-393.

40. Moreland, R.L. Transactive memory: Learning who knows what in work groups and organizations. *Shared Cognition in Organizations: The Management of Knowledge* Chapter 1, (1999), 3-31.

41. Moreland, R.L. and Myaskovsky, L. Exploring the performance benefits of group training: Transactive memory or improved communication. *Organizational Behavior and Human Decision Processes* 82, 1 (2000), 117-133.

42. Nelson, R.R., Todd, P.A., and Wixom, B.H. Antecedents of information and system quality: An empirical examination within the context of data warehousing. *Journal of Management Information Systems* 21, 4 (2005), 199-235.

43. Nidumolu, S.R., Subramani, M., and Aldrich, A. Situated learning and the situated knowledge web: Exploring the ground beneath knowledge management. *Journal of Management Information Systems* 18, 1 (2001), 115-150.

44. Ravichandran, T. and Lertwongsatien, C. Effect of information systems resources and capabilities on firm performance: A resource-based perspective. *Journal of Management Information Systems* 21, 4 (2005), 237-276.

45. Ravichandran, T. and Rai, A. Total quality management in information systems development: Key constructs and relationships. *Journal of Management Information Systems* 16, 3 (2000), 119-155.

46. Ruiz, F., Vizcaino, A., Piattini, M., and Garcia, F. An ontology for the management of software maintenance projects. *International Journal of Software Engineering and Knowledge Engineering* 14, 3 (2004), 1-27.

47. Schneidewind, N.F. Measuring and evaluating maintenance process using reliability, risk, and test metrics. *IEEE Transactions on Software Engineering* 25, 6 (1999), 769-781.

48. Shenhar, A.J. One size does not fit all projects: Exploring classical contingency domains. *Management Science* 47, 3 (2001), 394-414.

49. Swanson, E.B. Maintaining IS quality. *Information and Software Technology* 39, 12 (1997), 845-850.

50. Swanson, E.B. and Beath, C.M. Departmentalization in software development and maintenance. *Communications of the ACM* 33, 6 (1990), 658-667.

51. Van de Ven, A.H. Running in packs to develop knowledge-intensive technologies. *MIS Quarterly* 29, 2 (2005), 365-378.

52. Vessey, I. and Galletta, D.F. Cognitive Fit: An empirical study of information acquisition. *Information Systems Research* 2, 1 (1991), 63-84.

53. Wegner, D.M. Transactive memory: A contemporary analysis of the group mind. *Theories of Group Behavior*, (1987).

54. Wholey, D.R. and Brittain, J. Characterizing environmental variation. *Academy of Management Journal* 32, 4 (1989), 867-882.

55. Wu, X., Murray, A., Storey, M.-A., and Lintern, R.A Reverse engineering approach to support software maintenance: Version control knowledge extraction. *Proceedings of the Working Conference on Reverse Engineering*, 2004.

56. Xia, W. and Lee, G. Complexity of information systems development projects: Conceptualization and measurement development. *Journal of Management Information Systems* 22, 1 (2005), 45-83.

57. Xu, P. and Ramesh, B. Impact of knowledge support on the performance of software process tailoring. *Journal of Management Information Systems* 25, 3 (2008-2009), 277-314.

58. Yap, C.M., and Souder, W.E. Factors influencing new product success and failure in small entrepreneurial high-technology electronic firms. *Journal of Product Innovation Management* 11, 5 (1994), 418-432.

**Figure 1. Conceptual Model**



Figure 2. Moderating Effects of Software Volatility Patterns on the Relationship between Software Maintenance Approaches on Software Quality

|  |  |
| --- | --- |
| Software Volatility Pattern 1 | Software Volatility Pattern 2 |
|  |  |
| Software Volatility Pattern 3 | Software Volatility Pattern 4 |
|  |  |
| Vertical axis represents software quality (error rates). Lower scores represent higher software quality. |
| Blue: Technology-based approach | Red/Dotted: Experience-based approach | Green/Dashed: Skill-based approach |

Figure 3. Moderating Effects of Software Volatility Magnitude by Comparing Knowledge Approaches between Patterns 2 and 3

|  |  |
| --- | --- |
| Technology-based Approach | Skill-based Approach |
|  |  |
| Experience-based Approach |
|  |
| Vertical axis represents software quality (error rates). Lower scores represent higher quality. |
| Blue: Technology-based approach | Red: Experience-based approach | Green: Skill-based approach |
| Solid line: Volatility Pattern 2 | Doubled Line: Volatility Pattern 3 |

Table 1. Mapping of Software Volatility Patterns Based on Software Volatility Factors

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Software Volatility Factors | Pattern 1 (P1) | Pattern 2 (P2) | Pattern 3 (P3) | Pattern 4 (P4) |
| Modification frequency | High | X |  |  | X |
| Low |  | X | X |  |
| Modification predictability | High |  | X | X | X |
| Low | X |  |  |  |
| Modification magnitude | Large |  |  | X | X |
| Small | X | X |  |  |

Table 2. Summary of Variables

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Variable | Description | Mean | St. Dev. | Lag? | Transformed |
| *Dependent variable* |
| ERROR | Number of defects in nightly batch runs during time period t | 2.26 | 1.10 | No | Log |
| *Independent variables* |
| Volatility pattern | An application’s software volatility pattern, P2, P3, or P4. Pattern 1 is the base pattern. | NA | NA | NA | NA |
| TECH | Proportion of the application that was created or maintained with CASE tools | 0.26 | 0.27 | Yes | Normalized |
| EXP | Average number of days spent together by team members during time period t | 128.80 | 144.55 | Yes | Normalized |
| SKILL | Average skill of assigned maintainers for time period t | 3.49 | 1.12 | Yes | Normalized |
| *Control variables* |
| AGE | Application age | 122.55 | 59.22 | Yes | No |
| APPFP | Function points for the application | 2,290.60 | 1,501.83 | Yes | Normalized |
| LOC | Lines of code for the application | 356,757.4 | 33,7874.2 | Yes | Normalized |
| COMPL | Complexity measure (ZN2 per appl. module)  | 470.88 | 226.85 | Yes | Normalized |
| TXNUM | Number of online transactions in time period t | 642,837.4 | 1,001,172 | No | Normalized |

Table 3. Volatility Patterns Model Results

|  |  |
| --- | --- |
| Variable | Full Model |
| β # | Coefficient | p level |
| P2 | 1 | 0.7271 | .000 |
| P3 | 2 | 0.7000 | .000 |
| P4 | 3 | 1.1180 | .001 |
| TECH | 4 | -0.2910 | .000 |
| EXP | 5 | -0.2085 | .004 |
| SKILL | 6 | -0.4730 | .000 |
| TECH x P2 | 7 | -0.3944 | .002 |
| TECH x P3 | 8 | -1.2513 | .000 |
| EXP x P2 | 10 | 0.1184 | .218 |
| EXP x P3 | 11 | 0.2332 | .043 |
| EXP x P4 | 12 | 1.0439 | .006 |
| SKILL x P2 | 13 | 0.5821 | .010 |
| SKILL x P3 | 14 | 0.1364 | .324 |
| SKILL x P4 | 15 | 0.9824 | .000 |
| AGE | 16 | 0.0039 | .000 |
| APPFP | 17 | 0.0482 | .582 |
| LOC | 18 | 0.7435 | .000 |
| COMPL | 19 | 0.2083 | .000 |
| TXNUM | 20 | -0.1264 | .002 |
| Constant | 0 | 1.3159 | .000 |
|  |  |
| Wald’s χ2 |  | 20399.49 | 0.000 |

\*A potential TECH x P4 (β9) variable was dropped from the regression due to collinearity with TECH. The collinearity between the two variables was due to zero variation of TECH within the P4 volatility pattern.

Table 4. Summary of Coefficient Tests for Interactions

|  |  |  |
| --- | --- | --- |
| Volatility Pattern 1 |  | Volatility Pattern 2 |
|  | Tech | Exp |  |  | Tech | Exp |
| Exp | 0.0825(5.35)\*\*\* |  | Exp | -0.5128(26.46)\*\*\* |  |
| Skill | -0.1820(10.95)\*\*\* | -0.2645(15.78)\*\*\* | Skill | -0.9765(40.08)\*\*\* | -0.4637(14.60)\*\*\* |
| Volatility Pattern 3 |  | Volatility Pattern 4 |
|  | Tech | Exp |  |  | Tech | Exp |
| Exp | -1.4845(69.00)\*\*\* |  | Exp |  |  |
| Skill | -1.3877(62.00)\*\*\* | 0.0968(4.71)\*\*\* | Skill |  | 0.0615(2.014) \* |
| Score represents the difference between the associated coefficients, and the t value of the differential.Note: The interaction of TECH with P4 was dropped due to collinearity with TECH, thus no comparisons could be made with TECH for the Volatility Pattern 4 chart\* p < .05; \*\* p < .01; \*\*\* p < .001 |

Table 5. Hypotheses Results Summary

|  |  |  |
| --- | --- | --- |
| **#** | **Hypothesis** | **Supported?** |
| **1a** | When corrective software maintenance is more frequent, the effectiveness of the skill-based approach will be enhanced more than that of technology-based approaches in terms of reducing software errors | **Yes** |
| **1b** | When corrective software maintenance is more frequent, the effectiveness of the experience-based approach will be enhanced more than that of technology-based approaches in terms of reducing software errors | **No** |
| **2a** | When corrective software maintenance is less frequent, the effectiveness of technology-based approaches to knowledge sharing will be enhanced more than that of skill-based approaches, in terms of reducing software errors | **Yes** |
| **2b** | When corrective software maintenance is less frequent, the effectiveness of technology-based approaches to knowledge sharing will be enhanced more than that of experience-based approaches, in terms of reducing software errors | **Yes** |
| **3** | When corrective software maintenance is more unpredictable, the effectiveness of skill-based approaches to knowledge sharing will be enhanced more than the experience-based approach, in terms of reducing software errors | **Yes** |
| **4a** | When corrective software maintenance is of a greater magnitude, a technology-based approach to knowledge sharing will be enhanced in terms of reducing software errors when compared to maintenance of a smaller magnitude | **Yes** |
| **4b** | When corrective software maintenance is of a greater magnitude an experience-based approach to knowledge sharing will be enhanced in terms of reducing software errors when compared to maintenance of a smaller magnitude | **No** |
| **4c** | When corrective software maintenance is of a larger magnitude, the effectiveness of a skill-based approach to knowledge sharing will be enhanced in terms of reducing software errors when compared to maintenance of a smaller magnitude | **Yes** |

1. Given the nature of time series data, it is appropriate to lag variables by one time interval in order to mitigate the effects of possible endogeneity [30]. For example, see [25]. Our study uses a monthly time interval. [↑](#footnote-ref-1)
2. To account for differences in this variable that may be due to the age of the system, this measure is adjusted by dividing it by the application’s age. [↑](#footnote-ref-2)
3. For more information on this tool, please refer to: <http://en.wikipedia.org/wiki/CA-Telon> [↑](#footnote-ref-3)
4. Meaning that each value of the variable was subtracted from its mean and then divided by its standard deviation [↑](#footnote-ref-4)
5. We note that our model uses the first volatility pattern (P1) as the base case, and that the effects of the other volatility patterns (P2, P3, and P4) are in relation to the P1 case. [↑](#footnote-ref-5)
6. Application volatility pattern P1 is the “base” case, so the test simply involves comparing the coefficients on the variables for TECH, EXP and SKILL. Application volatility pattern P4 also has a high frequency of modification, but technology is not used for the applications with this pattern, so it is not possible to test the hypothesis using this pattern. [↑](#footnote-ref-6)
7. Contrary to our expectation, the experience-based approach was slightly more error-prone in comparison to the technology-based approach (H1b). A similar finding was reported by Lewis [36], in that frequent communication among team members, diminished the effectiveness of their memory recall due to the increase in coordinated placed on the team by the frequent interactions. It is possible that frequent maintenance efforts likewise significantly increase the coordination load upon the team and thereby diminish its ability to quickly and accurately recall information needed to maintain and produce high quality software. [↑](#footnote-ref-7)
8. In fact, as suggested in Figure 3 the experience-based approach appears to be equally effective for both large and small modifications. [↑](#footnote-ref-8)