9-2011

Structural complexity and programmer team strategy: An experimental test

Narayan RAMASUBBU
Singapore Management University

Chris F. Kemerer

Jeff Min Teck HONG
Singapore Management University

DOI: https://doi.org/10.1109/TSE.2011.88

Published in IEEE Transactions on Software Engineering, 2011 September, Volume 38, Issue 5, Pages 1054-1068
https://doi.org/10.1109/TSE.2011.88

Structural Complexity and Programmer Team Strategy: An Experimental Test

Narayan Ramasubbu, Chris F. Kemerer, *Member*, *IEEE Computer Society*, and Jeff Hong

Abstract—This study develops and empirically tests the idea that the impact of structural complexity on perfective maintenance of object-oriented software is significantly determined by the team strategy of programmers (independent or collaborative). We analyzed two key dimensions of software structure, coupling and cohesion, with respect to the maintenance effort and the perceived ease-of-maintenance by pairs of programmers. Hypotheses based on the distributed cognition and task interdependence theoretical frameworks were tested using data collected from a controlled lab experiment employing professional programmers. The results show a significant interaction effect between coupling, cohesion, and programmer team strategy on both maintenance effort and perceived ease-of-maintenance. Highly cohesive and low-coupled programs required lower maintenance effort and were perceived to be easier to maintain than the low-cohesive programs and high-coupled programs. Further, our results would predict that managers who strategically allocate maintenance tasks to either independent or collaborative programming teams depending on the structural complexity of software could lower their team's maintenance effort by as much as 70 percent over managers who use simple uniform resource allocation policies. These results highlight the importance of achieving congruence between team strategies employed by collaborating programmers and the structural complexity of software.

Index Terms—Object-oriented programming, complexity measures, software quality, software productivity, programming teams, maintenance process, CK metrics, software management

1 Introduction

Organizations spend a significant proportion of their IT budgets on software maintenance with aims to improve software quality and to prolong system life. However, a disproportionate allocation of resources to maintenance activities can potentially reduce the ability of firms to innovate through new application development, a phenomenon termed the "legacy crisis" [1].

In response to the challenge of reducing system maintenance costs, a wide range of techniques have been developed by the software engineering research community [2], [3], [4]. A fundamental principle often utilized by these techniques is that software maintenance is strongly influenced by structural complexity, i.e., the manner in which program elements are organized within a system [5], [6]. It has been shown that through better design the interconnections between the various elements of a system can be simplified to aid maintainability [5], [7], [8], [9], [10]. However, a majority of the research investigating the relationship between software structure and maintenance has either been conducted 1) pertaining to an individual maintainer's approach to maintenance (e.g., cognition and program comprehension studies [11], [12]), or 2) has addressed the software structure without detailed attention to programmers' strategies (e.g., complexity metrics studies [2], [13], [14], [15]).

While both of these factors (programmer strategy and software structure) have influence on the final outcome, the interactions of these two elements have generally been neglected, which leaves open the possibility that simply better matches of programmer strategies and situations may result in improved performance outcomes. In addition, there has been a consistent and growing emphasis on team approaches to software development and maintenance in both commercial software development and in software engineering education [16], [17], [18], [19], [20], [21], [22], [23]. Therefore, there is a need to study the relationship between systems maintenance and system structure in more detail by accommodating the programmer team strategies which influence the conduct of system maintenance activities in order to determine if there are complementary team mechanisms for specific software structures. Expanding the unit of analysis to include both the software structural elements and the human factors also presents an opportunity to bridge the prescriptions offered by both the program comprehension and the software complexity research streams, and has the opportunity to positively influence maintenance management practice. The objective of the study is to take a step forward in this direction by examining the joint impact of object-oriented software structure and programmer team strategies on software maintenance. The study also offers a contribution to the growing use of experimental design in empirical software engineering research.

Variations in programmer team strategies during software maintenance are typically caused by the different

- *N. Ramasubbu and C.F. Kemerer are with the Joseph M. Katz Graduate School of Business, University of Pittsburgh, 276C Mervis Hall, Pittsburgh, PA 15260. C.F. Kemerer is also with King Abdul Aziz University, Saudi Arabia. E-mail: {narayanr, ckemerer}@katz.pitt.edu.*
- *J. Hong is with the School of Information Systems, Singapore Management University, 80 Stamford Road, Singapore 178902. E-mail: minteck.2010@smu.edu.sg.*

Manuscript received 28 July 2010; revised 30 June 2011; accepted 15 Aug. 2011; published online 22 Aug. 2011.
Recommended for acceptance by M.-A. Storey.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-2010-07-0237.
Digital Object Identifier no. 10.1109/TSE.2011.88.

ways in which teams achieve their division of labor. Two widely used team strategies in software projects are independent team programming and collaborative team programming [16], [17], [18], [19], [20], [21], [22], [23]. *Independent team programming* (hereafter simply "independent programming") refers to a team strategy where maintenance tasks are divided among programmers who work in parallel on separate parts of a system and coordinate their efforts [16], [23]. *Collaborative team programming* (hereafter simply "collaborative programming") refers to a scheme where two or more programmers work together on the same parts of a system, rather than working in parallel on two different parts of the system [16], [18]. It has been shown in organizational studies that the efforts required to achieve mutual understanding of a problem and to coordinate among team members under alternative team strategy regimes can differ significantly [25], [26], [27], and therefore the outcomes for independent versus collaborative programming strategies can be expected to also differ.

Past research studies examining programmer team strategies (including pair programming) have not explicitly accounted for the possible joint effects of system complexity and team strategy in their design [16], [17], [18], [19], [20], [21], [23].[1] This study investigates maintenance tasks done by pairs of programmers and specifically focuses on the differences between two different team strategies—independent programming and collaborative programming—employed by the pairs of programmers, and how the interaction between the software structure and the team strategy influences maintenance performance. The central research question answered by this study is this: *What is the effect of programmer team strategy on software maintenance performance for different levels of structural complexity?*

To answer this research question we conducted a controlled lab experiment with 45 professional programmer pairs (90 subjects). We found that programmers' maintenance effort levels (in person minutes) and ease-of-maintenance perceptions for the two different team strategies were highly contingent upon the structural complexity levels that they encountered. In the lowest possible structural complexity environment of the experiment, programmers employing the independent programming strategy required 49 percent less effort than programmers employing the collaborative programming strategy. But, in environments with higher structural complexity levels, teams using the collaborative programming strategy required from a minimum of 12 percent to a maximum of 51 percent less effort than teams using the independent programming strategy in the same complexity settings. Further, programmers' perceptions of ease-of-maintenance for modules with low-structural complexity were approximately 30 percent higher than more structurally complex modules, and all else being equal, the collaborative programming strategy was perceived to be easier to use (approximately 28 percent higher) than the independent programming strategy. The results of this study highlight how the programmer team strategies and the structural complexity of a system can interact to jointly influence maintenance performance variables such as maintenance effort and ease-of-maintenance perceptions.

The remainder of this paper is organized as follows: The theoretical background on the key constructs of this study is presented in Section 2. Research hypotheses are developed in Section 3. The empirical research design to test the hypotheses and the experimental procedures are described in Section 4. Section 5 presents the analysis of the data and the results of the hypothesis tests. Section 6 discusses the results and their limitations, and Section 7 concludes the paper by highlighting the contributions of this study.

2 Theoretical Background

We used two main theoretical perspectives for hypothesis development. We conceptualized the properties of software maintenance tasks undertaken by collaborating programmers using the *distributed cognition theoretical framework* [28]. And we analyzed the impact of the team strategy employed by the programmers and its impacts on maintenance performance using the *task interdependence theoretical framework* [25], [27], [29].

2.1 Distributed Cognition Framework

Software maintenance is recognized both as a cognitive activity dependent on an individual programmer's system comprehension and as a social activity involving frequent interactions between programmers working in teams [30], [31]. The distributed cognition framework posits the study of such cognitive phenomena by taking into account the social context in which the actors are situated, and treating the actors, their interactions with one another, and their environment as a single distributed cognitive system [28], [32].

Flor and Edwin [24] were among the pioneers in the application of the distributed cognition framework to the study of software maintenance activities. Rogers and Ellis [33] detailed the theoretical basis of distributed cognition for studying collaborative activity. Other researchers have also utilized the distributed cognition framework to study pair programming teams [34]. Collectively, the stream of literature examining the application of the distributed cognition framework to study software maintenance teams recommends the analysis of the following properties:

1. structure and frequency of tasks,
2. team structure and the coordination mechanisms used,
3. tools, documents, and the patterns of use of these artifacts,
4. development of shared knowledge.

These properties derived from the distributed cognition framework are utilized for the design of this study. We treated a pair of programmers and the software application they worked on as a distributed cognition system. We then observed the activities of the programmer pairs, their team strategies (work division and coordination mechanisms), and performance under different environments of structural complexity where appropriate tool usage was experimentally controlled.

1. Although, see Arisholm and Sjoberg [47] for a recent notable exception which uses control delegation (centralized versus decentralized) as a complexity measure. Our study proposes a contingency view of structural complexity and utilizes standard coupling and cohesion object-oriented metrics to measure complexity.

2.2 Task Interdependence Framework and Team Strategies

Task interdependence is the degree to which team members must rely on each other to perform their tasks [25], [27], [29]. Prior research in organizational studies and psychology has shown that increased task interdependence is associated with increased requirements for coordination and communication effort among team members in order to perform their tasks well [25], [27]. Different modes of task interdependence have been categorized based on the information and workflow sequences between team members performing the tasks. Under *reciprocal interdependence*, team members perform different parts of a task in flexible order as per their specializations, and then coordinate among themselves using temporally lagged, two-way interactions to complete the whole task [27], [29]. In contrast, under *team interdependence*, team members jointly diagnose, solve problems, and collaborate to perform a task. Unlike reciprocal interdependence, team interdependence involves simultaneous work interactions and requires group discretion for interactions between team members. In our study, one pair of programmers performing a maintenance task had the opportunity to work in parallel and independently, and this group, referred to as following the independent programming strategy, corresponds to the theoretical "reciprocal interdependence" classification. Another group (pair of maintainers) worked jointly to complete a maintenance task, and is referred to as following the collaborative programming strategy; this group corresponds to the theoretical "team interdependence" classification.[2]

By mapping the programmer team strategies to the theoretical task interdependence classifications, we can build upon the insights from past psychology and organization theory research studies which have shown that the congruence between the nature of a task and the interdependence of team members can significantly impact group performance and the perceived effectiveness of team members, and that the selection of appropriate team strategies often requires a careful assessment of the information processing and coordination demands of team tasks [26], [27], [35], [36]. In prior research, structural complexity of software has been used to examine and assess the information processing and coordination demands placed on team members working on a maintenance task [5]. In this study, we extend the analysis by assessing the congruency between the structural complexity of software and the team strategy employed by programmers.

2.3 Structural Complexity of Software

There is a rich body of the software engineering literature associating the structural properties of systems with their maintainability [2], [4], [5], [6], [37], [38], [39]. Early studies adopted specific characteristics of programming languages, such as usage of long jumps, GO TO statements, depth of IF statement hierarchies, etc., for characterizing structure, and this early work has been generalized to focus on coupling and cohesion as the key measurable conceptual properties of the structural complexity of software.

Coupling represents the interdependencies between software elements in a system, and *cohesion* captures the similarities or binding of elements that are grouped together [5]. Several metrics have been developed for coupling and cohesion for both procedural and object-oriented designs, and are described in detail in prior research [40], [41], [42], [43], [44], [45]. Automated tools to gather coupling and cohesion metrics from existing software systems are commercially available as well [46]. A majority of research studies that have analyzed the impact of coupling and cohesion on higher order measures of software quality and productivity have concluded that low coupling and high-cohesion designs generally yield systems of higher quality that are easier to maintain [5], [47]. Further, maintenance effort has been shown to be influenced by the interaction between coupling and cohesion, implying the advantage of considering coupling and cohesion jointly, not merely independently, in software system design decisions [5].

2.4 Maintenance Performance

Maintenance performance of a programming team has been typically assessed in prior studies by measuring maintenance effort, i.e., the total person minutes spent by a programming team to complete a maintenance task [5], [13], [48]. In our study, we follow a similar plan by measuring effort spent by programmers on a perfective maintenance task to assess team maintenance performance. In addition to maintenance effort, our research design also calls for capturing programmers' perceptions of the ease-of-maintenance using the chosen team strategy. People's subjective beliefs on "ease of doing" have been shown to act as influential behavioral determinants of accepting technology and processes irrespective of their inherent objective qualities [49], [50], [51]. Therefore, it is important to assess programmers' perceptions of ease-of-use of maintenance team strategies, along with other objective measures of performance, such as maintenance effort, to assess the relevance of different team strategies while planning resource allocation policies and project work breakdown structures. Further, an understanding of the programmer's perceptions on ease-of-maintenance under different team strategies can help managers formulate policies that aid organization-wide assimilation and acceptance of favorable team strategies. Similarly to perceived ease-of-use measures employed in prior technology acceptance studies (e.g., [49], [50], [51]), perceived ease-of-maintenance is defined as a programming team's subjective appraisal (on Likert scales of 1-5 ($1 = \text{hardest}$ to perform maintenance, $5 = \text{easiest}$ to perform maintenance)) of the ease of conducting the maintenance tasks under the specific team strategy that was assigned to them.

3 Hypotheses

3.1 Coupling, Cohesion, and Maintenance Effort

The main effects of coupling and cohesion on maintenance effort are well documented (e.g., [5]), and our first set of hypotheses is designed to check those results in our

2. In the organizational literature there are other forms of task interdependence. Under *pooled interdependence*, each team member makes a contribution to team output without the need for any direct interaction with other team members; *sequential task interdependence* requires specific behaviors to be performed by team members in a predetermined order [27], [29]. These theoretical forms of interdependence are not applicable in our research context, and thus are not considered further here.

experimental setting and to establish a baseline to compare against later results.

A highly *cohesive* system binds similar software elements in a single place, and is expected to aid program comprehension by minimizing a software maintainer's search and exploration tasks. These benefits are expected to translate to higher order gains in the form of lowered maintenance effort. However, a highly *coupled* system has relatively many interconnections between its software elements, which hinder program comprehension. Programmers working on highly coupled systems need to carefully explore a typically large array of possible interconnections when making changes to individual software elements, necessitating a higher relative maintenance effort. Further, highly coupled systems are not easy to "chunk" into logical information processing units due to the large number of interconnections, which hinders the learning process of maintenance personnel, potentially leading to higher maintenance effort expenditure [5].

Program comprehension by human actors may depend upon both coupling and cohesion rather than simply their individual effects. Following Darcy et al. [5], it is also expected that a significant interaction effect may exist between coupling and cohesion that impacts maintenance effort. Thus, the first set of confirmatory hypotheses is

- H1. Maintenance effort is lower for the more highly cohesive programs.
- H2. Maintenance effort is higher for the more highly coupled programs.
- H3. For more highly coupled programs, maintenance effort is lower if cohesion levels are high.[3]

3.2 Programmer Team Strategy and Maintenance Effort

Two specific strategies for programmer teamwork, independent programming and collaborative programming, are considered in this study. Under the independent programming strategy, maintenance tasks are "split and conquered" among team members, enabling parallel work. But, when looked at from a task interdependence point of view, independent programming entails additional effort expenditure on explicit coordination (boundary spanning activities) to synchronize parallel work among team members [22], [52]. Moreover, additional effort has to be spent on achieving common ground when team members deal with errors at the boundary of their individual work that ripples across in different directions.

Under the collaborative programming strategy, team members jointly work on all activities and do not have to spend as much effort on boundary spanning activities. However, the savings that stem from parallel work are not possible under the collaborative programming strategy. Thus, the final relative performance of independent programming and collaborative programming with regard to maintenance effort is likely to depend upon other factors that influence coordination and comprehension effort. For this research, we view maintenance activity as a distributed cognition system, with the maintainers and the system as intertwined elements, and therefore posit that the structural complexity of the software needs to be considered to differentiate the effects of independent and collaborative programming on maintenance performance.

Following the logic of the first set of hypotheses, maintenance effort is expected to be relatively lower in low coupling/high-cohesion environments. As established in prior research [5], it is expected that, in such low-structural complexity regimes, coordination and boundary spanning overhead efforts between programmers working on a maintenance task are lower. Thus, savings arising from parallel work enabled by the independent programming strategy would outweigh the costs of overhead efforts (coordination and boundary spanning) associated with it. However, such savings are not possible under the collaborative programming strategy because it does not facilitate parallel work between collaborating programmers. Therefore, in low-structural complexity regimes (high cohesion/low coupling), we expect that maintenance effort of independent programming strategy will be lower than that of collaborative programming strategy.

In contrast, in high-structural complexity environments (high coupling/low cohesion) where achieving higher levels of program comprehension is generally harder, it is expected that the coordination and boundary spanning overhead costs of independent programming will outweigh the costs of collaboration programming (lack of parallel work). Therefore, we expect the maintenance effort of collaborative programming strategy to be lower in higher structural complexity regimes.

In summary, we expect the levels of cohesion, coupling, and their interactions with the chosen team strategy (independent or collaborative programming) to significantly determine effort spent on a maintenance task. Hypotheses related to the three-way interaction between coupling, cohesion, and team strategy can be stated in any combination of lower/higher levels of each of the three interacting variables. For simplicity, we explicitly enumerate only the following combinations among the three-way interaction as our second set of hypotheses:

- H4. For the more highly cohesive programs, the independent programming strategy will be associated with lower relative maintenance effort.
- H5. For the more highly coupled programs, the collaborative programming strategy will be associated with lower relative maintenance effort.
- H6. Under the collaborative programming team strategy for the more highly coupled programs, maintenance effort will be lower if cohesion levels are high.
- H7. Under the independent programming team strategy for the more highly cohesive programs, maintenance effort will be higher if coupling levels are high.

3.3 Task Strategy and Perceived Ease

Prior research on antecedents of perceived ease-of-use shows that individuals use "anchoring and adjustment" heuristics to form their decisions on ease-of-use [53], [54], [55]. Initial anchoring might be based on an individual's prior knowledge and inherent beliefs, and adjustment to the

3. The interaction hypothesis can also be stated in several equivalent alternative ways in terms of characterizing the different levels of coupling and cohesion. For simplicity, we explicitly state only one of the possible combinations in the two-way interaction between coupling and cohesion.

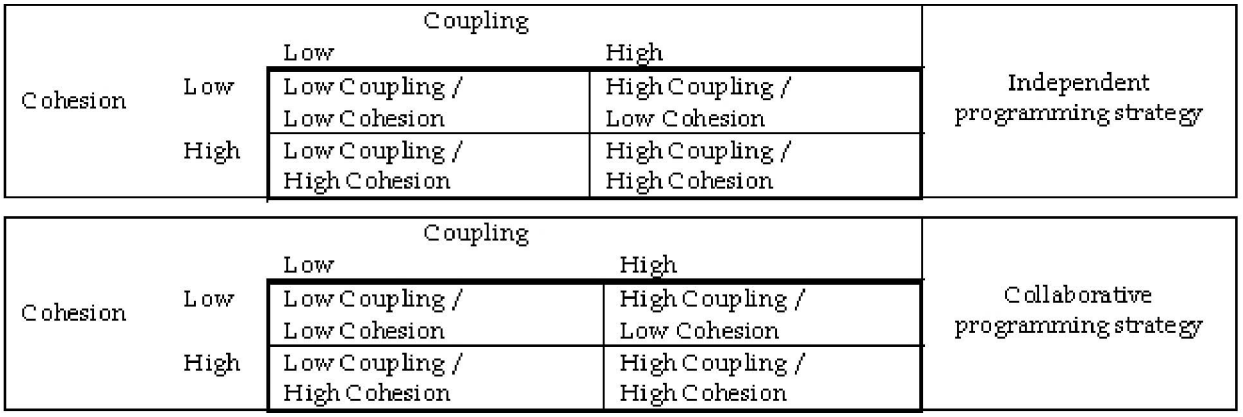

Fig. 1. Experiment design.

initial anchor of perceived ease-of-use is often influenced by the social contexts of an individual's task environment. Formal training, informal learning, and knowledge transfer through group interactions serve as important facilitating conditions for adjustments to initial anchors of perceptions on ease of doing a task [53]. Thus, in the context of software maintenance teamwork, all else held equal, the ease with which team members are able to interact with and learn from each other influences programmers' perceived ease-of-system maintenance.

It is expected that the collaborative programming strategy would be perceived as more easy to use than the independent task strategy because collaborative programming facilitates the development of communal and "sharing-the-burden perceptions" through explicit joint-work processes. Under the collaborative programming strategy, programmers jointly conduct diagnosis and problem solving activities, and can learn from each other. Since such group interactions in collaborative programming are built into the regular work process, programmers do not experience an extra burden for knowledge transfer. In contrast, under the independent programming strategy, programmers encounter an additional burden to coordinate and exchange knowledge, which could be expected to dampen the formation of positive ease-of-use perceptions. Thus, our next hypothesis is:

H8. The perceived ease-of-maintenance of the collaborative programming task strategy will be higher than the perceived ease-of-maintenance of the independent programming task strategy.

Following the logic of interaction effects posited in our first and second set of hypotheses, it is also likely that perceptions on ease-of-maintenance for independent and collaborative programming strategies are varied according to the structural complexity regimes encountered by the programmers. Especially, the interdependence between code elements as represented by coupling can be expected to influence programmer perceptions on ease-of-maintenance along with their task interdependence due to their chosen team strategy. Since coupling acts as a dampening mechanism for easy interactions between collaborating programmers, we can expect the positive ease-of-maintenance perceptions typically engendered by a collaborative programming strategy to decrease in higher coupling environments. Therefore, our next hypothesis is:

H9. The difference in perceived ease-of-maintenance between the collaborative programming task strategy and the independent programming task strategy will decrease with an increase in the level of coupling.

4 Research Design and Experiment Procedures

4.1 Experiment Design

A controlled lab experiment method was chosen to collect data for testing the hypotheses. A $2 \times 2 \times 2$ between subjects experiment design, as shown in Fig. 1, was chosen with the following three factors: 1) coupling (low-high), 2) cohesion (low-high), and 3) team strategy (independent programming and collaborative programming), which generates eight (2^3) possible conditions.

The dependent variables were maintenance effort and perceived ease-of-maintenance. As described above, maintenance effort was measured in person-minutes and perceived ease-of-maintenance as the average score of a three item interview questionnaire with responses sought on 5-point Likert scales. Responses from the team members were sought through an interview on 1) ease of understanding the business logic of the system while working on the maintenance task, 2) ease of understanding the technical design and operation of the system while working on the maintenance task, and 3) overall ease of performing the maintenance task under the assigned team strategy. Coupling and cohesion were measured using two CK object-oriented software metrics (coupling using CBO, cohesion using LCOM) [43]. These specific object-oriented metrics to measure coupling and cohesion were chosen as their properties with respect to software maintenance are well documented in prior research [5], [15], [43], [46], providing a sound basis for comparing our experimental results with prior published structural complexity experiments, especially [5].

4.2 Experiment System, Manipulation of Factors, and Subject Tasks

A stable version of an existing database and reporting application system written in Java (Java SE 6, update 11) was chosen as the primary experiment artifact. The

application had 13,400 lines of code, 20 database tables, 185 SQL statements, and 85 interfaces among the various classes and Java Server Pages. In preparation of the experiment artifacts an extensive manual walkthrough of the system source code was conducted, along with an object-oriented metrics extraction using the CKJM tool [56].

Four different versions of the system with the same business functionality, but with varying levels of coupling and cohesion corresponding to the experiment design, were developed from the original application. The refactoring techniques we used to manipulate coupling and cohesion were motivated by prior published structural complexity experiments [5], [9], [10]. Coupling was primarily manipulated by modifying method calls, and cohesion was primarily manipulated by adjusting the sharing of instance variables between method pairs. For example, when a method is used by more features of another class than the class on which it is defined, we created a new method with a similar body in the class where it is used most, and we then either converted the old method into a simple delegation, or removed it altogether. We also converted local variables to fields—if the local variable is initialized on creation, then the refactoring operation moves the initialization to the new field's declaration or to the class's constructors. These refactoring manipulations yielded two distinct levels of cohesion (low, $\mathrm{LCOM} = 45$; high, $\mathrm{LCOM} = 10$) and coupling (low, $\mathrm{CBO} = 6$; high, $\mathrm{CBO} = 12$). The manipulations did not significantly alter the code size (measured in lines of code (LOC)) of the application (the differences in LOC among the four versions was less than 1 percent). Two independent computer science PhD-holding experts (not the authors) who were familiar with the system usage and had more than 10 years of object-oriented software development expertise were provided with the source code of the four different artifacts corresponding to our experiment design and were asked to verify and report the equivalence in business functionality and accuracy of the metrics collection. The experts confirmed that the four different versions of the system had the same functionality and depicted different structural complexity levels as measured by the CBO and LCOM coupling and cohesion metrics. The level of interrater reliability as measured by Cohen's Kappa of the verification exercise was 0.9, indicating a high degree of consensus between the raters.

We designed a perfective maintenance task to be completed by all subjects (pairs of professional programmers). The context of the perfective maintenance task was that the organization using the system had instituted a new operational location, and a subset of its operational activities was to be run at the new location. The design of the perfective maintenance task was motivated by a real-world case drawn from an observation of the development requests reported for the system, which improves the ecological validity of our experiment. Subjects were asked to modify the system in order to accommodate the new user requirement. The same IDE (JEdit), test data, and sample reports were provided to all subjects.

4.3 Pretest and Power Analysis

We pretested the experimental system and planned procedures and conducted a pilot study with two pairs of professional programmers and four pairs of advanced university students majoring in information systems. We conducted a power analysis using the data collected from the pilot study to estimate the sample size required for the experiment design. Similarly to past software engineering research [48], we chose the desired power for the model as 0.8 where the effect size was based on the task completion rate, cell means, and standard deviations from the pretest data, with the alpha set to 0.05. The power analysis indicated that approximately 40 pairs, or 80 programmers, were needed for the $2 \times 2 \times 2$ fixed effect experiment design to appropriately test all main effects and interactions.

4.4 Subjects

In order for the research results to have substantial external validity to commercial environments, eligibility to serve as an experimental subject was limited to professional programmers with a minimum of 2 years of Java programming language experience and possessing an official "Java Programmer" certification [57]. Volunteer programmers were solicited through a professional special interest group on Java programming in Singapore, the site of the experiment. E-mail advertisements for the experiment were also sent through the human resources divisions of three leading software services firms located in Singapore. Ultimately, 45 pairs of programmers, or 90 subjects, participated in the experiment.

4.5 Procedures

Pairing of programmers and subject (pair) allocation to experimental cells was done randomly. When the subjects arrived on site they were briefed about the experiment, a high-level overview of the experimental system was presented, and two training tasks were given. The training tasks were different from the main experiment tasks, but were designed to help the subjects become familiarized with the different modules of the application. All subjects received identical training.

Subjects were required to work on laptops provided for the experiment which had identical hardware configurations and installed applications. For subjects in the independent programming strategy group, two laptops were provided for each pair, whereas only one laptop was provided for subjects in the collaborative programming group. The laptops were preloaded with the appropriate variant of the experiment application, test data, and sample reports, and screen capture recording software. The screen capture software was used to track the exact timing of maintenance events. Subjects were required to check in their completed code to a version control system. Once subjects indicated task completion, tests were run on their final checked-in version to determine the accuracy of their solution. If errors were found, the subjects were notified and asked to rectify the errors. Only when the solutions passed all of the acceptance tests was the submission deemed complete. The time required for solution validation by the supervisor was not counted as part of the maintenance effort. Upon completion of the task, subjects completed a postexperiment interview and were compensated 25 SGD for their participation in the experiment. All subjects completed the experiment within the planned 2 hours, and there were no dropouts.

TABLE 1
Descriptive Statistics for Potential Covariates

Covariate	Units	Min	Max	Mean	Std. Dev
Age	Years	21	26	23.88	1.70
Java Experience	Years	2	4	2.40	0.58
Programming Career Experience	Years	2	8	3.72	1.55
Number of programming languages known	Absolute number	2	5	3	1.17
Undergraduate GPA	Absolute number	3.19	3.9	3.46	0.30

Throughout the experiment an observer was present in the lab along with the subjects. The observer kept track of the experiment time (start and end of comprehension activities, coordination activities, and execution activities), documented the work division between programmers, and made nonintrusive general observations of the task progress. The experiment observations were corroborated with data from the screen capture videos and check-in, check-out patterns from the version control system. The three way check of experimental data from observer notes, screen capture videos, and the version control system served to minimize any measurement-related human errors.

5 Analysis and Results

5.1 Data Analysis

We analyzed the experiment data using version 11 of the STATA statistical package [58]. In the first stage, we verified the normal distribution of the response variables, maintenance effort, and perceived ease-of-maintenance, using the Shapiro-Wilk test [59] and through visual inspection of QQ plots, skewness, and kurtosis graphs [60]. These tests did not reveal any normality-related issues. An outlier analysis was performed to check for potentially influential or erroneous outliers. This analysis revealed two candidate cases. In one of the cases maintenance effort was lower than the respective cell mean and in the other case higher. We checked all data on the two candidate cases and found no errors, and therefore we retained these cases in the data set (the final results are robust to both the inclusion and the exclusion of the two candidate outlier cases).

Descriptive statistics of the potential covariates collected through the postexperiment interview and their correlations with the response variables are shown in Tables 1 and 2, respectively. None of the potential covariates was significantly correlated with either maintenance effort or perceived ease-of-maintenance. We verified the homogenous distribution of covariates across the eight experiment cells through a series of Analysis of Variance (ANOVA) tests with the covariates as dependent variables, and coupling, cohesion, and team strategy as the independent variables. None of these ANOVA models was statistically significant, implying homogenous distribution across the experiment cells.

Since the research model of this study involved two response variables (maintenance effort and perceived ease-of-maintenance), we performed a Multivariate Analysis of Variance (MANOVA). Table 3 shows the results of this analysis. The overall model was statistically significant (Table 3, Row 1, $\mathrm{F} = 2.95$, $p\text{-value} = 0$), confirming that there were significant differences in the means of maintenance effort and perceived ease-of-maintenance across the different experiment cells. Referring to Table 3 it can be seen that the main effects of cohesion (Table 3, Row 2) and coupling (Table 3, Row 3) are both highly significant at the usual levels. The two-way interaction between coupling and cohesion (Table 3, Row 4) is also significant in the overall model at the $p < 0.10$ level. The independent main effect of team strategy was not significant at usual levels (Table 3, Row 5), but the two-way and three-way interactions of team strategy with coupling and cohesion were found to be statistically significant (Table 3, Rows 6-8), indicating that the interaction effect of team strategy and structural complexity of software is a significant driver of performance outcomes.

We also performed separate univariate Analysis of Variance (ANOVA) analyses for both maintenance effort and perceived ease-of-maintenance. The univariate models were statistically significant, and results of these univariate models were similar to the MANOVA analysis (maintenance effort model: $\mathrm{F} = 5$, $p\text{-value} = 0.001$, adj. $R\text{-squared} = 0.39$; perceived ease-of-maintenance model: $\mathrm{F} = 2.9$, $p\text{-value} = 0.01$, adj. $R\text{-squared} = 0.23$).

TABLE 2
Correlations for Maintenance Effort and Perceived Ease-of-Use with Potential Covariates†

Variables	Age	Java Experience	Programming Career Experience	Number of programming languages	Undergraduate GPA
Maintenance Effort	0.14 (0.30)	-0.04 (0.79)	-0.03 (0.86)	0.13 (0.38)	-0.09 (0.58)
Perceived Ease-of-Maintenance	0.24 (0.11)	0.09 (0.54)	-0.13 (0.41)	-0.08 (0.6)	0.11 (0.46)

Note: † *P-Values in parenthesis.*

TABLE 3
MANOVA for Maintenance Effort and Perceived Ease-of-Maintenance

		Pillai's trace statistic#	F-statistic	p-value
1	MANOVA Model (Adj. R^2 = 0.39; n=45)	1.03	2.95	0.00
2	Cohesion	0.28	4.57	0.01
3	Coupling	0.24	3.78	0.02
4	Cohesion*Coupling	0.18	2.61	0.07
5	Team Strategy	0.04	0.5	0.68
6	Cohesion*Team Strategy	0.19	2.72	0.06
7	Coupling*Team Strategy	0.17	2.36	0.09
8	Cohesion*Coupling*Team Strategy	0.21	3.08	0.04

Note: # The results of the MANOVA analysis were identical across different test statistics (Pillai's trace, Wilk's lambda, Roy's largest root, and Lawley-Hotelling trace.

TABLE 4
Hypothesis Tests Results

No.	Hypothesis	Observed Difference	Statistically Supported? †	Statistical Test Result†
Maintenance Effort Hypotheses				
H1	Maintenance effort is lower for the more highly cohesive programs.	High Cohesion 47% less effort than Low Cohesion	Yes	Chi-squared=11.93, P=0.000***
H2	Maintenance effort is higher for the more highly coupled programs.	High Coupling 80% higher effort than Low Coupling	Yes	Chi-squared=10.43, P=0.000***
H3	For the more highly coupled programs, maintenance effort is lower if cohesion levels are high.	[High Coupling/High Cohesion] 56% less effort than [High Coupling/Low Cohesion]	Yes	Chi-squared=19.10, P=0.000***
H4	For the more highly cohesive programs, independent programming strategy is associated with lower maintenance effort.	Independent programming 11% less effort than Collaborative programming	No	Chi-squared=0.08, P=0.77
H5	For the more highly coupled programs, collaborative programming strategy is associated with lower maintenance effort.	Collaborative programming 12% less effort than Independent programming	No	Chi-squared=1.02, P=0.31
H6	Under the collaborative programming strategy, for the more highly coupled programs, maintenance effort is lower if cohesion levels are high.	[High Coupling/High Cohesion] 63% less effort than [High Coupling/Low Cohesion]	Yes	Chi-squared=40.43, P=0.000***
H7	Under the independent programming strategy, for the more highly cohesive programs, maintenance effort is higher if coupling levels are high.	[High Cohesion/High Coupling] 122% higher effort than [High Cohesion/Low Coupling]	Yes	Chi-squared=10.94, P=0.000***
Perceived Ease-of-Maintenance Hypotheses				
H8	The perceived ease-of-maintenance of the collaborative programming task strategy will be higher than the perceived ease-of-maintenance of the independent programming task strategy.	Perceived ease-of-maintenance of Collaborative programming is 28% higher than Independent programming	Yes	Chi-squared=8.35, P=0.004***
H9	The difference in perceived ease-of-maintenance between the collaborative programming task strategy and the independent programming task strategy will decrease with an increase in the level of coupling.	[Collaborative programming – Independent programming] for High Coupling scored 86% lower on perceived ease- of-maintenance than [Collaborative programming – Independent programming] for Low Coupling	Yes	Chi-squared=10.40, P=0.000***

Note: † All p-values are two-tailed; Bonferroni adjusted p-values are 0.006 for 5 percent significance (marked as "" in Table 4).*

5.2 Hypotheses Tests

We examined the individual hypotheses developed in Section 2 using posthoc tests following the MANOVA analysis. Since all the hypothesis tests comparisons were done using the same MANOVA results, a Bonferroni adjustment was applied to the p-values to minimize Type 1 errors [62], and these results are shown in Table 4. The observed differences (reported in Table 4, column 3) between the hypothesized comparison conditions are calculated using the experiment cell means, which are

TABLE 5
Experiment Cell Means Data—Maintenance Effort†

		Independent Programming	Collaborative Programming
1	Low Cohesion and Low Coupling	41 (2.73)	19.83 (1.32)
2	Low Cohesion and High Coupling	70.2 (4.68)	61.67 (4.11)
3	High Cohesion and Low Coupling	15 (0.98)	29.33 (1.96)
4	High Cohesion and High Coupling	33.23 (2.15)	24.67 (1.64)

Note: † Standard deviation in parentheses; maintenance effort is measured in person-minutes.

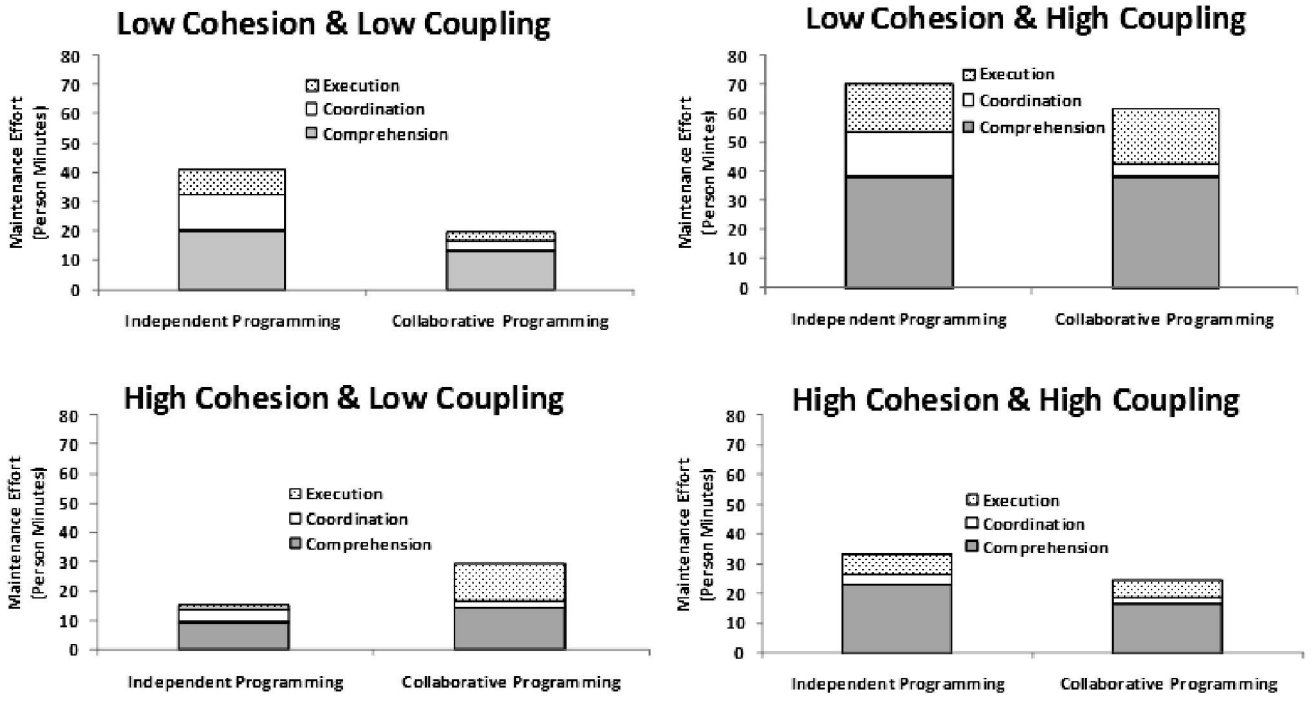

Fig. 2. Effects of coupling, cohesion, and task strategy on maintenance effort.

TABLE 6
Experiment Cell Means Data—Perceived Ease-of-Maintenance†

		Independent Programming	Collaborative Programming
1	Low Cohesion and Low Coupling	2.5 (0.11)	4 (0.17)
2	Low Cohesion and High Coupling	3 (0.13)	3.7 (0.16)
3	High Cohesion and Low Coupling	2.25 (0.09)	4.5 (0.19)
4	High Cohesion and High Coupling	3.7 (0.16)	3.7 (0.14)

Note: † Standard deviation in parentheses; perceived ease-of-maintenance measured on a Likert scale ranging from 1-5 (1 = hardest to perform maintenance, 5 = easiest to perform maintenance).

reported in Table 5 and Fig. 2 for maintenance effort and Table 6 and Fig. 3 for perceived ease-of-maintenance.

The statistical tests reported in Table 4, column 5, verify if each of the observed differences are significant at the Bonferroni adjusted $\alpha = .006$ level. For example, the value for H1 in Table 4 is calculated as follows: Total maintenance effort for the low-cohesion condition is 192.7 as derived using the cell means of the low-cohesion condition $(= 41 + 19.8 + 70.2 + 61.7 = 192.7)$. Similarly, total maintenance effort for high-cohesion condition is derived as 102.2 $(15 + 29.3 + 33.2 + 24.7 = 102.2)$. Calculating the ratio [(high cohesion-low cohesion)/low cohesion] as a percentage (i.e., $[(102.2 - 192.7)/192.7] * 100 = -46.96$), shows that the highly cohesive programs require about 47 percent lower maintenance effort than the low-cohesive programs. This observed difference is statistically significant as shown by the Chi-squared statistic in Table 4 H1 (p-value $= 0.000$).

All of the confirmatory hypotheses for the maintenance effort (H1, H2, and H3) were supported. Maintenance effort was lower for highly cohesive programs, higher for highly coupled programs, and there was a significant interaction effect between coupling and cohesion in determining maintenance effort.

While we found significant interaction effects for team strategy in the MANOVA analysis (Table 3, Rows 6-8), a comparison of means as posited by hypotheses H4 and H5 did not reveal statistically significant results at the Bonferroni adjusted $\alpha = 0.006$ level. Even though the observed differences between High-Cohesion and Low-Cohesion groups under the independent programming strategy (refer

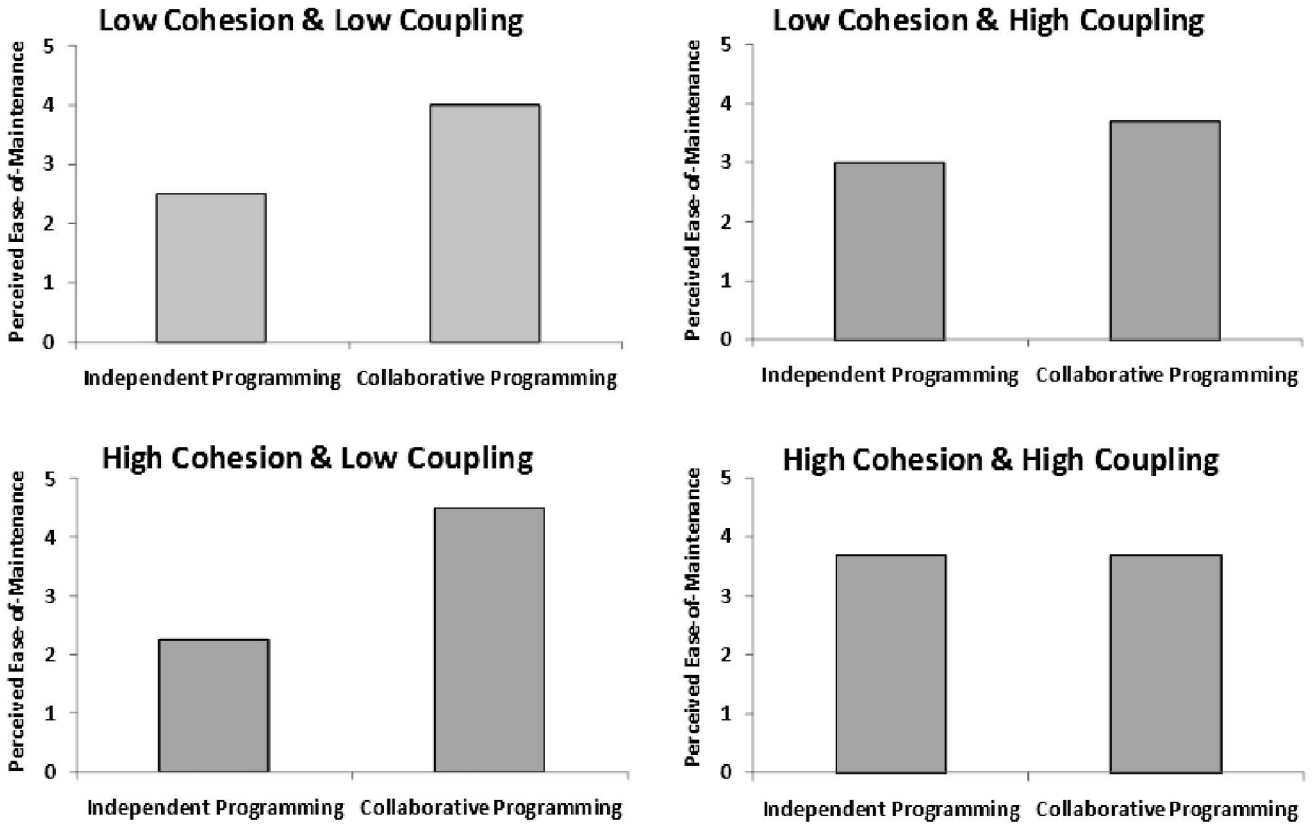

Fig. 3. Effects of coupling, cohesion, and team strategy on perceived ease-of-maintenance.

to Table 4, H4) and the High-Coupling and Low-Coupling groups under the collaborative programming strategy (refer to Table 4, H5) are in the hypothesized directions, they are not statistically significant at the more conservative level. This indicates that we cannot confirm how the impact of structural complexity on performance outcomes is influenced by a chosen team strategy by only considering one of coupling or cohesion. Rather, there is a need to consider the full three-way interaction effects between coupling, cohesion, and team strategy in order to examine how the interplay among these variables impacts maintenance performance. Hypotheses H6 and H7, which proposed a three-way interaction between coupling, cohesion, and task strategy, were both strongly supported (p-value $= 0.000$, refer to Table 4, H6, H7).

Fig. 2 visually shows the differences in cell means of maintenance effort for all the groups in the research design (three-way between coupling, cohesion, and team strategy). Fig. 2 also includes the subdivision of overall maintenance effort into program comprehension, explicit coordination, and execution portions. These subdivisions of maintenance effort were derived based on the events noted by the experiment observer and was corroborated using the screen capture recordings and version control system check-in timings. The three-way interaction effects between coupling, cohesion, and team strategy on maintenance effort are visually represented in Fig. 4.

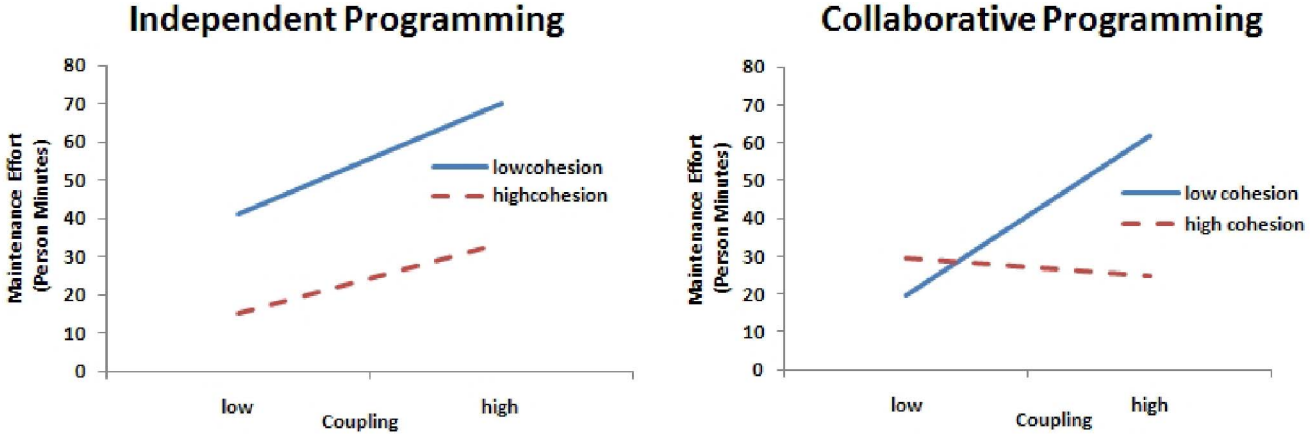

Fig. 4. Interaction effects of coupling, cohesion, and task strategy on maintenance effort.

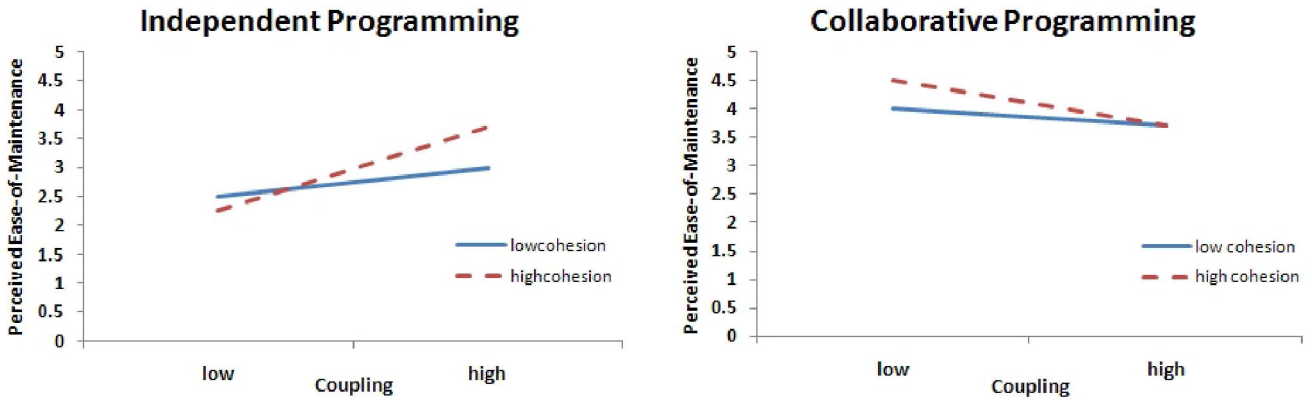

Fig. 5. Interaction effects of coupling, cohesion, and team strategy on perceived ease-of-maintenance.

Both of the perceived ease-of-maintenance hypotheses (refer to Table 4, H8, and H9) were fully supported at the Bonferroni adjusted $\alpha = 0.006$ level. As we had expected, the perceived ease-of-maintenance level for the collaborative programming strategy groups was higher than for the independent programming strategy groups. Also, as hypothesized in H9, under the collaborative programming strategy highly coupled programs showed lower perceived ease-of-maintenance levels than programs with lower levels of coupling. The perceived ease-of-maintenance levels for the various experimental groups are presented in Fig. 3. The significant three-way interaction between coupling, cohesion, and task strategy is depicted visually in Fig. 5.

It is interesting to note that, under the independent programming team strategy, programmers tended to perceive maintenance of highly coupled programs as easier than less coupled programs, even though it takes them more effort to complete the maintenance of highly coupled programs. We believe that this result is driven by their difficulty in establishing an initial common ground between collaborating programmers in higher structural complexity contexts. This is supported by the observed higher levels of program comprehension effort expended by the programmers for highly coupled programs (see Fig. 2). In Section 6.1, we discuss the implications of this type of potential mismatch between what may be programmers' preferred choice and what can be shown to be the economically optimal team strategy.

6 Discussion

The primary objective of this research was to extend the investigations of the relationship between software structure and maintenance performance by taking into account the team strategies employed by maintenance programmers. Supported by the theoretical perspectives of the distributed cognition and task interdependence frameworks, this study experimentally validated that the team strategy employed by maintenance teams, along with structural complexity, are important factors in influencing performance outcomes such as maintenance effort and perceived ease-of-maintenance.

Specific differences in maintenance effort across different levels of structural complexity and team strategies can be inferred from Fig. 2 and Table 5, which show the cell means of all the combinations of the interacting variables. Since the perfective maintenance task (i.e., adding a business functionality) across the experiment cells remained constant, the observed differences in maintenance effort can be interpreted as productivity differences[4] induced due to the congruence (or lack thereof) of team strategies and software structure.

Referring to Fig. 2, it can be seen that, other than in the high-cohesion/low-coupling quadrant (lowest structural complexity), teams using the collaborative programming strategy were more productive (required less total effort) than teams using the independent programming strategy, *ceteris paribus*. The largest difference in productivity between the collaborative and independent programming strategies can be seen in the low-cohesion/low-coupling quadrant (49.5 percent), and the smallest difference is found in the low-cohesion/high-coupling quadrant (highest structural complexity) (14 percent). However, in the lowest possible structural complexity environment of the experiment (high-cohesion/low-coupling quadrant), programmers employing the independent programming strategy were 50.2 percent more productive on average than the programmers employing the collaborative team strategy. Irrespective of the team strategies employed, maintenance of high-cohesion programs was 47 percent more productive than maintenance of low-cohesion programs. Similarly, it required on average 80 percent more effort from programmers to finish maintenance tasks in highly coupled programs as compared to programs with lower levels of coupling.

Referring to Fig. 5, it can be seen that programmers' ease-of-maintenance perceptions for the team strategies were also highly contingent on the structural complexity levels that they encountered. Programmers' perception of ease-of-maintenance for modules with high cohesion and low coupling were 30 percent higher than other more complex modules, and all else being equal, the collaborative programming strategy was perceived to be easier to use (28 percent higher) than the independent programming strategy. However, the ease-of-maintenance perception difference between collaborative programming and independent programming dropped significantly (86 percent) as coupling increased.

4. That is, the numerator in the equivalent productivity equation (output/input = maintenance task size/effort) remained the same across the cells.

Due to the limited sample size of our professional programmer pool, we performed statistical tests on theoretically motivated hypotheses pertinent only to a subset of all the possible combinations of the $2 \times 2 \times 2$ experiment design reported in this paper. However, it is also possible to glean interesting qualitative results from Figs. 2 and 3 on the untested effects. For example, from Fig. 2 we can see that under the collaborative programming strategy, the effects of coupling on maintenance effort is reversed when cohesion changes from low to high. That is, collaborative programming becomes a viable team strategy in high-coupling regimes if accompanied by high cohesion. Similar reversal of effects with respect to perceived ease-of-maintenance can be observed from Fig. 3. For example, while the independent programming strategy typically yields the best ease-of-maintenance scores, once high coupling is accompanied by high cohesion, independent programming no longer outweighs the collaborative programming strategy.

Other qualitative results can be derived by observing the variety of descriptive differences in lower order factors of program comprehension, coordination, and solution execution effort due to the dynamic interactions between software structure and team strategy. We plotted these patterns in Fig. 2 using a combination of version control check-in and screen capture data along with the unintrusive observations of experiment observers. The pattern of breakdowns in maintenance effort show that program comprehension and execution effort are typically higher in more structurally complex environments, but the coordination effort needed to complete a maintenance task is more dependent on the team strategy employed by the collaborators. While coordination effort is generally lower for the collaborative programming strategy, by employing the independent programming strategy in less structurally complex environments it may be possible to exploit the lower effort needed for solution execution to boost maintenance performance.

These results provide evidence for the proposition that managers should take a contingency view of structural complexity when planning maintenance projects. When the maintenance activity of teams is viewed as a distributed cognitive system, the impacts of structural complexity are not determined by the structure of the software alone, but are contingent on the team strategies that are employed by the software maintainers. Referring to Figs. 2 and 3, one can see how the results of the programmer team strategy contingency of structural complexity provides different maintenance effort levels and perceived ease-of-maintenance for different groups of the interaction.

6.1 Implications for Research and Practice

Software engineering research studies that compare different processes or techniques (e.g., individual versus pair programming) often view structural complexity as a static function and merely control for its effect by including levels of coupling and cohesion (or other similar metrics) in their models. Instead, based on the results of this study, a more nuanced view of structural complexity is advocated. This study shows that when examining higher order factors such as productivity, it is necessary to account for how maintainers involved in the software activity approach the inherent software structure and act on it. Our study provides a rationale using the distributed cognition and task interdependence frameworks to support a contingency view of software complexity, and experimentally validated the use of team strategy-coupling-cohesion interactions as a way to account for effects of software complexity on maintenance performance. Team strategy has the benefit of being a controllable manifest dimension of team structure, and could be particularly useful when the unit of analysis in software research is a team or project, rather than an individual.

An important implication of the contingency view of structural complexity for practice is on the way work breakdown structures are achieved in a software project. This research study shows that there are higher order benefits, including improved maintenance productivity, which could be reaped if careful attention is paid to achieve congruence between project task work breakdown structures and team strategies employed by programmers. Work breakdown structures capture the planned division of labor mechanism by managers. Programmer team strategies determine how the elements of a work breakdown structure are further chunked, clustered, and performed during the actual task execution. Hence, incongruence between the planned work breakdown structures and the actual team strategies may result in unexpected overhead costs, such as unplanned coordination and idle time.

Our study shows that significant improvements in maintenance performance can be attained if the latent structural properties of object-oriented systems are exploited for project planning, deriving work breakdown structures, and resource allocation. For example, our results would predict that managers who allocate maintenance tasks to independent or collaborative programming teams depending on the structural complexity of software (e.g., high-cohesion/low-coupling maintenance to independent programming teams and more complex tasks to collaborative programming teams) could lower their team's maintenance effort by as much as 70 percent over managers who use a simple uniform resource allocation policy. It is important to note, however, that choice of the maintenance team strategy might be affected by programmers' willingness to employ it, and managers should be aware that programmers might prefer a team strategy different than the economically optimal one. Thus, in order to deploy an optimal resource allocation policy derived from the contingency view of structural complexity, additional complementary investments in, for example, training programs and team-building exercises, might be necessary.

It is possible to discover the latent structural properties of object-oriented systems at relatively low cost by using commercially available object-oriented metrics and toolsets. Therefore, this study further suggests the value of integrating object-oriented metrics into the early stage project planning process. However, getting leading indicators of software structural complexity through object-oriented metrics can sometimes be challenging in practice, due to customer restrictions, or to the lack of implementation of automated tools. One way to break such a deadlock is through local tailoring of processes and through treating team strategy as a

response to a given software structure that is being discovered concurrently. A more refined, metrics-driven strategy of allowing independent programmers to team program on high cohesion/low coupled program elements and collaboratively pairing programmers to handle low cohesion/highly coupled program elements could potentially result in significant savings in total effort expended.

6.2 Limitations and Future Research

This research study is based on a controlled experiment using pairs of certified professional programmers with at least 2 years of commercial experience. Although high confidence can be placed in the specific results due to the use of experimental controls, normal caution has to be exercised on broad generalizations. Nevertheless, since the hypotheses of this study are theoretically motivated, the procedures can be easily replicated in other empirical settings and the results verified.

To be able to control the important factors of interest and to keep the sample size feasible given the use of professional programmers, some other potentially interesting variables observed in field settings were not considered as part of the research design. While keeping the research model parsimonious helped in maintaining control of the primary factors of interest, such designs necessitate tradeoffs with other potential research questions. For example, this experiment leaves to future research possible manipulation of programmer expertise (e.g., novices versus senior programmers) or variations in team sizes. Future research could extend the findings of this study to corrective and adaptive maintenance, and also to study the impact of structural complexity on other potentially relevant response variables, such as reuse or conformance quality.

The $2 \times 2 \times 2$ experiment design we used for the study did not lend itself to also considering a broad range of refactoring techniques available for manipulating structural complexity of software and, unlike other studies [9], [10], we did not specifically focus on the effects induced by specific refactoring techniques on programmer behavior. We restricted our refactoring actions to manipulate coupling and cohesion by modifying method calls and the sharing of instance variables between method pairs. This limits the development of finer prescriptions on how a system with given structural complexity can be altered to suit a preferred team strategy of a project team. We believe that this does not pose a serious threat to the validity of our results since we ensured functional equivalence of the various versions of the system and also used a completely randomized assignment for the experiment groups. However, we caution against broad generalizability until the results reported in this study have been replicated using a variety of refactoring techniques. Assessing the sensitivity of the three-way interaction effects of coupling, cohesion, and team strategies with respect to refactoring techniques is a potentially fruitful area of future research.

Also, as is traditional with software engineering experiments, we acknowledge the limitations of generalizing our findings to large maintenance projects that operate for extended periods of time. And normal caution has to be placed on extending laboratory experimental results based on a relatively small perfective maintenance task to larger production settings. Replication and verification of the results using data from larger and diverse production environments is recommended.

Given that the current study has shown the effect of two consistent team strategies, independent and collaborative programming, future research could investigate the possible effects of mixed or hybrid task strategy models. Finally, the impact of different modes of pairing programmers in collaborative programming setting (experts-novices, novices-novices, experts-experts) on the three-way interaction between team strategy, coupling, and cohesion could be examined in future research as well.

7 Conclusion

This study provides evidence establishing the relationship between the structure of systems and maintenance performance by accommodating the nature of work division mechanisms employed by maintenance teams. Viewing the combination of the system and the system maintainers as intertwined components of a single distributed cognitive system, a contingency view of structural complexity is established. Using data collected from a controlled lab experiment with professional programmer pairs as subjects the contingency view of structural complexity is illuminated by demonstrating the presence of interactions between the structural properties (coupling and cohesion) of the system and team strategies of the actors (independent programming versus collaborative programming). The key finding of the experiment is that the latent structural properties of object-oriented systems can be exploited to improve maintenance performance by appropriately choosing between independent programming and collaborative programming strategies. Maintenance effort and perceived ease-of-maintenance of programmers are significantly influenced by the complex three-way interactions between coupling, cohesion, and task strategy. This study provides an empirically validated rationale for using the coupling-cohesion-team strategy framework for planning maintenance projects and for resource allocation. The wide availability of object-oriented metrics and tool sets provides ample impetus to accomplish this in software engineering practice.

Acknowledgments

Helpful comments on earlier drafts were received from Sherae Daniel, David Darcy, Lingxiao Jiang, Sandra Slaughter, Kevin Steppe, Giri Kumar Tayi, Jason Woodard, the associate editor, and three anonymous referees.

References

[1] R.C. Seacord, D. Plakosh, and G.A. Lewis, *Modernizing Legacy Systems*. Addison-Wesley, pp. 1-16, 2003.

[2] C.F. Kemerer, "Software Complexity and Software Maintenance: A Survey of Empirical Research," *Annals of Software Eng.*, vol. 1, pp. 1-22, 1995.

[3] J.T. Nosek and P. Palvia, "Software Maintenance Management: Changes in the Last Decade," *J. Software Maintenance and Evolution: Research and Practice*, vol. 2, pp. 157-174, 2006.

[4] N.F. Schneidewind, "The State of Software Maintenance," *IEEE Trans. Software Eng.*, vol. 13, no. 3, pp. 303-310, Mar. 1987.

[5] D.P. Darcy, C.F. Kemerer, S.A. Slaughter, and J.E. Tomayko, "The Structural Complexity of Software: An Experimental Test," *IEEE Trans. Software Eng.*, vol. 31, no. 11, pp. 982-995, Nov. 2005.

[6] R.S. Sangwan, P. Vercellone-Smith, and P.A. Laplante, "Structural Epochs in the Complexity of Software over Time," *IEEE Software,* vol. 25, no. 4, pp. 66-73, July/Aug. 2008.

[7] E. Arisholm, "Empirical Assessment of the Impact of Structural Properties on the Changeability of Object-Oriented Software," *Information and Software Technology,* vol. 48, pp. 1046-1055, 2006.

[8] R.D. Banker and S.A. Slaughter, "The Moderating Effects of Structure on Volatility and Complexity in Software Enhancement," *Information Systems Research,* vol. 11, pp. 0219-0240, 2000.

[9] B. Du Bois, S. Demeyer, and J. Verelst, "Does the 'Refactor to Understand' Reverse Engineering Pattern Improve Program Comprehension?" *Proc. Ninth European Conf. Software Maintenance and Reeng.,* 2005.

[10] B. Du Bois, S. Demeyer, and J. Verelst, "Refactoring—Improving Coupling and Cohesion of Existing Code," *Proc. 11th Working Conf. Reverse Eng.,* 2004.

[11] M.A. Storey, "Theories, Tools and Research Methods in Program Comprehension: Past, Present and Future," *Software Quality J.,* vol. 14, pp. 187-208, 2006.

[12] A.v. Mayrhauser and A.M. Vans, "Identification of Dynamic Comprehension Processes during Large Scale Maintenance," *IEEE Trans. Software Eng.,* vol. 22, no. 6, pp. 424-437, June 1996.

[13] R.K. Bandi, V.K. Vaishnavi, and D.E. Turk, "Predicting Maintenance Performance Using Object-Oriented Design Complexity Metrics," *IEEE Trans. Software Eng.,* vol. 29, no. 1, pp. 77-87, Jan. 2003.

[14] F. Fioravanti and P. Nesi, "Estimation and Prediction Metrics for Adaptive Maintenance Effort of Object-Oriented Systems," *IEEE Trans. Software Eng.,* vol. 27, no. 12, pp. 1062-1084, Dec. 2001.

[15] R. Subramanyam and M.S. Krishnan, "Empirical Analysis of CK Metrics for Object-Oriented Design Complexity: Implications for Software Defects," *IEEE Trans. Software Eng.,* vol. 29, no. 4, pp. 297-310, Apr. 2003.

[16] J.T. Nosek, "The Case for Collaborative Programming," *Comm. ACM,* vol. 41, pp. 105-108, 1998.

[17] A. Parrish, R. Smith, D. Hale, and J. Hale, "A Field Study of Developer Pairs: Productivity Impacts and Implications," *IEEE Software,* vol. 21, no. 5, pp. 76-79, Sept./Oct. 2004.

[18] L. Williams, R.R. Kessler, W. Cunningham, and R. Jeffries, "Strengthening the Case for Pair Programming," *IEEE Software,* vol. 17, no. 4, pp. 19-25, July/Aug. 2000.

[19] V. Balijepally, R. Mahapatra, S. Nerur, and K.H. Price, "Are Two Heads Better Than One for Software Development? The Productivity Paradox of Pair Programming," *MIS Quarterly,* vol. 33, pp. 91-118, 2009.

[20] N. Salleh, E. Mendes, and J. Grundy, "Empirical Studies of Pair Programming for CS/SE Teaching in Higher Education: A Systematic Literature Review," *IEEE Trans. Software Eng.,* vol. 37, no. 4, pp. 509-525, July/Aug. 2010.

[21] C. McDowell, L. Werner, H.E. Bullock, and J. Fernald, "Pair Programming Improves Student Retention, Confidence, and Program Quality," *Comm. ACM,* vol. 49, pp. 90-95, 2006.

[22] S. Sawyer, "Software Development Teams," *Comm. ACM,* vol. 47, pp. 95-99, 2004.

[23] K.M. Lui, K.C.C. Chan, and J. Nosek, "The Effect of Pairs in Program Design Tasks," *IEEE Trans. Software Eng.,* vol. 34, no. 2, pp. 197-211, Mar./Apr. 2008.

[24] N.V. Flor and H. Edwin L, "Analyzing Distributed Cognition in Software Teams: A Case Study of Team Programming During Perfective Software Maintenance," *Proc. Fourth Workshop Empirical Studies of Programmers,* pp. 36-64, 1991.

[25] D.J. Campbell, "Task Complexity: A Review and Analysis," *Academy of Management Rev.,* vol. 13, pp. 40-52, 1988.

[26] R. Wageman, "Interdependence and Group Effectiveness," *Administrative Science Quarterly,* vol. 40, pp. 145-180, 1995.

[27] R. Saavedra, P.C. Earley, and L.V. Dyne, "Complex Interdependence in Task-Performing Groups," *J. Applied Psychology,* vol. 78, pp. 61-72, 1993.

[28] J. Hollan, E. Hutchins, and D. Kirsch, "Distributed Cognition: Towards a New Foundation for Human-Computer Interaction Research," *ACM Trans. Computer-Human Interaction,* vol. 7, pp. 174-196, 2000.

[29] J.D. Thompson, *Organizations in Action: Social Science Bases of Administrative Theory.* Transaction Publishers, 2003.

[30] R.E. Kraut and S. Lynn A, "Coordination in Software Development," *Comm. ACM,* vol. 38, pp. 69-81, 1995.

[31] C. Bird, D. Pattison, R. D'Souza, V. Filkov, and P. Devanbu, "Latent Social Structure in Open Source Projects," *Proc. 16th ACM SIGSOFT Foundations of Software Eng.,* 2008.

[32] E. Hutchins, *Cognition in the Wild.* MIT Press, 1996.

[33] Y. Rogers and J. Ellis, "Distributed Cognition: An Alternative Framework for Analysing and Explaining Collaborative Working," *J. Information Technology,* vol. 9, pp. 119-128, 1994.

[34] H. Sharp and H. Robinson, "A Distributed Cognition Account of Mature XP Teams," *Lecture Notes in Computer Science,* P. Abrahamsson, M. Marchesi, and G. Succi, eds., vol. 4044, pp. 1-10, Springer, 2006.

[35] J.E. McGrath, *Groups Interaction and Performance.* Prentice-Hall, 1984.

[36] P.S. Goodman, "Impact of Task and Technology on Group Performance," *Designing Effective Workgroups,* P.S. Goodman ed., pp. 120-167, Jossey-Bass, 1986.

[37] V.R. Gibson and J.A. Senn, "System Structure and Software Maintenance Performance," *Comm. ACM,* vol. 32, pp. 347-358, 1989.

[38] J. Hagemeister, B. Lowther, P. Oman, X. Yu, and W. Zhu, "An Annotated Bibliography on Software Maintenance," *ACM SIGSOFT Software Eng. Notes,* vol. 17, pp. 79-84, 1992.

[39] R.D. Banker, S.M. Datar, C.F. Kemerer, and D. Zweig, "Software Complexity and Maintenance Costs," *Comm. ACM,* vol. 36, pp. 81-94, 1993.

[40] N.E. Fenton and A.A. Kaposi, "Metrics and Software Structure," *Information and Software Technology,* vol. 29, pp. 301-320, 1987.

[41] R. Adamov and L. Richter, "A Proposal for Measuring the Structural Complexity of Programs," *J. Systems and Software,* vol. 12, pp. 55-70, 1990.

[42] H. Dhama, "Quantitative Models of Cohesion and Coupling in Software," *J. Systems and Software,* vol. 29, pp. 65-74, 1995.

[43] S.R. Chidamber and C.F. Kemerer, "A Metrics Suite for Object Oriented Design," *IEEE Trans. Software Eng.,* vol. 20, no. 6, pp. 476-493, June 1994.

[44] E. Arisholm, L.C. Briand, and F. Audun, "Dynamic Coupling Measurement for Object-Oriented Software," *IEEE Trans. Software Eng.,* vol. 30, no. 8, pp. 491-506, Aug. 2004.

[45] L.C. Briand, J.W. Daly, and J. Wust, "A Unified Framework for Cohesion Measurement in Object-Oriented Systems," *Empirical Software Eng.,* vol. 3, pp. 65-117, 1998.

[46] D.P. Darcy and C.F. Kemerer, "OO Metrics in Practice," *IEEE Software,* vol. 22, no. 6, pp. 17-19, Nov./Dec. 2005.

[47] E. Arisholm and D.I.K. Sjoberg, "Evaluating the Effect of a Delegated versus Centralized Control Style on the Maintainability of Object-Oriented Software," *IEEE Trans. Software Eng.,* vol. 30, no. 8, pp. 521-534, Aug. 2004.

[48] E. Arisholm, H. Gallis, T. Dyba, and D.I.K. Sjoberg, "Evaluating Pair Programming with Respect to System Complexity and Programmer Expertise," *IEEE Trans. Software Eng.,* vol. 33, no. 2, pp. 65-86, Feb. 2007.

[49] F.D. Davis, "Perceived Usefulness, Perceived Ease-of-Use, and End User Acceptance of Information Technology," *MIS Quarterly,* vol. 13, pp. 318-339, 1989.

[50] M.G. Morris and A. Dillon, "How User Perceptions Influence Software Use," *IEEE Software,* vol. 14, no. 4, pp. 58-65, July 1997.

[51] C.K. Riemenschneider, B.C. Hardgrave, and F.D. Davis, "Explaining Software Developer Acceptance of Methodologies: A Comparison of Five Theoretical Models," *IEEE Trans. Software Eng.,* vol. 28, no. 12, pp. 1135-1145, Dec. 2002.

[52] C.B. Seaman and V.R. Basili, "Communication and Organization: An Empirical Study of Discussion in Inspection Meetings," *IEEE Trans. Software Eng.,* vol. 24, no. 7, pp. 559-572, July 1998.

[53] V. Venkatesh, "Determinants of Perceived Ease of Use: Integrating Control, Intrinsic Motivation, and Emotion into the Technology Acceptance Model," *Information Systems Research,* vol. 11, pp. 342-365, 2000.

[54] V. Venkatesh and F.D. Davis, "A Model of the Antecedents of Perceived Ease of Use: Development and Test," *Decision Sciences,* vol. 27, pp. 451-481, 1996.

[55] A. Tversky and D. Kahneman, "Judgment under Uncertainty: Heuristics and Biases," *Science,* vol. 185, pp. 1124-1130, 1974.

[56] D.D. Spinellis http://www.spinellis.gr/sw/ckjm/, 31, May, 2010.

[57] Sun-Java, http://education.oracle.com/pls/web_prod-plq-dad/db_pages.getpage?page_id=320, 31, May, 2010.

[58] StataCorp "Stata Statistical Software: Release 11," College Station, StataCorp LP, 2009.
[59] S.S. Shapiro and M.B. Wilk, "An Analysis of Variance Test for Normality," *Biometrika*, vol. 52, pp. 591-611, 1965.
[60] X. Chen, P. Ender, M. Mitchell, and C. Wells, "Regression with Stata," http://www.ats.ucla.edu/stat/stata/webbooks/reg/default.htm, July 2010.
[61] C.J. Huberty and S. Olejnik, *Applied MANOVA and Discriminant Analysis*. Wiley-Interscience, 2006.
[62] J.P. Shaffer, "Multiple Hypothesis Testing," *Ann. Rev. Psychology*, vol. 46, pp. 561-584, 1995.

Narayan Ramasubbu received the bachelor's of engineering degree from Bharathiar University, India, and the PhD degree from the University of Michigan, Ann Arbor. He is an assistant professor at the Katz Graduate School of Business at the University of Pittsburgh. Previously, he was an assistant professor at the School of Information Systems at the Singapore Management University. Prior to his academic career, he was a senior developer at SAP AG and CGI Inc. His current research interests include software engineering economics with a focus on globally distributed product development and service delivery; the design, implementation, and governance of enterprise information systems; end-user interaction and user-led innovation; and IT strategy and generating business value from IT.

Chris F. Kemerer received the BS degree from the Wharton School at the University of Pennsylvania and the PhD degree from Carnegie Mellon University. He is the David M. Roderick Chair in Information Systems at the Katz Graduate School of Business at the University of Pittsburgh. Previously, he was an associate professor at MIT. His research on management issues in information systems and software engineering and the adoption and diffusion of information technologies have led him to be named as an ISI/Thomson Reuters highly cited researcher in computer science. His senior editorial positions included serving as the editor-in-chief of *Information Systems Research* and as the departmental editor for information systems at *Management Science*. He is a member of the IEEE Computer Society and is a past associate editor of the *IEEE Transactions on Software Engineering*.

Jeff Hong received the BS degree in information systems management (magna cum laude) from the School of Information Systems at the Singapore Management University (SMU). He is currently working toward the PhD degree at SMU. Prior to joining SMU, he was part of the IT operational team of a multinational private bank and also started his own IT solutions company. He is a certified project management professional (PMP). His research interests include behavioral aspects of information systems management.