Effects of Structural Complexity and Team Task Strategy on Object-Oriented Software Maintenance: An Experimental Test

Narayan Ramasubbu

School of Information Systems
Singapore Management University
80 Stamford Road
Singapore 178902
nramasub@smu.edu.sg

Chris F. Kemerer

Joseph M. Katz School of Business
University of Pittsburgh
276C Mervis Hall, Pittsburgh, PA 15260
ckemerer@katz.pitt.edu

Jeff Hong

School of Information Systems
Singapore Management University
80 Stamford Road
Singapore 178902
mintec.2005@sis.smu.edu.sg

Keywords: collaborative programming, independent programming, cohesion, coupling, software maintenance effort, productivity, experiment, CK metrics, perfective maintenance.

Acknowledgments: Helpful comments on earlier drafts were received from David Darcy, Sherae Daniel, Sandra Slaughter, Giri Kumar Tayi, Kevin Steppe, and Jason Woodard.
Effects of Structural Complexity and Team Task Strategy on Object-Oriented Software Maintenance: An Experimental Test

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 task strategy of programming teams (independent or collaborative). Two key dimensions of software structure, coupling and cohesion, were analyzed with respect to both the maintenance effort and the perceived ease-of-use 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 task strategy on both maintenance effort and perceived ease-of-use. Maintenance of the highly cohesive programs required 47% lower effort than the low cohesive programs, and was 80% lower for programs with low coupling than for the highly coupled programs. Further, our results would predict that managers who allocate maintenance tasks to independent or collaborative programming teams depending on the structural complexity of software could lower their team’s maintenance effort by as much as 70% over managers who use simple uniform resource allocation policies. These results highlight the importance of achieving congruence between task strategies employed by collaborating team members and the structural complexity of software. (199 words)
1. Introduction

Organizations spend a significant proportion of their IT budgets on software maintenance with aims to improve system 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 has been developed by the software engineering research community [2-4]. 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]. However, a majority of the research investigating the relationship between software structure and maintenance has either been conducted (a) pertaining to an individual maintainer’s approach to maintenance (*e.g.*, cognition and program comprehension studies [9, 10]), or (b) has addressed the software structure without detailed attention to programmers’ maintenance approaches (*e.g.*, complexity metrics studies [2, 11-13]). While both of these factors (maintainer approach and software structure) have influence on the final outcome, the *interactions* of these two elements have generally been neglected and leaves open the possibility that better matches of approaches and situations may result in improved managerial 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 [14-21]. Therefore, there is a need to study the relationship between systems maintenance and system structure in more detail by accommodating the 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 the program comprehension and software complexity research streams, and has the opportunity to positively influence maintenance management practice. The objective of this study is to take a step in this direction by examining the joint impact of object-oriented software structure and task strategies of teams on software maintenance effort and perceived ease-of-use. The study also offers a contribution to the growing use of experimental design in empirical software engineering research.

Variations in task strategies of maintenance programmers are caused by the different ways in which teams achieve their division of labor. Two widely used task strategies in software projects are *independent team programming* and *collaborative team programming* [14-21]. *Independent team programming* (hereafter simply “independent programming”)\(^1\) refers to a task strategy where system maintenance tasks are divided among programmers who work in parallel on separate parts of the system and coordinate their efforts [14, 22]. On the other hand, *collaborative team programming* (hereafter simply “collaborative programming”) refers to a scheme where two or more programmers work together on the same piece of software rather than working in parallel on two different parts of the system [14, 16]. 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 task strategy regimes can be significantly different [23-25], and therefore, the outcomes for independent versus collaborative programming task strategies can be expected to also differ. The objective of this study is to investigate the joint impacts of software structure and task strategies on software maintenance. Past research studies examining collaborative programming generally have not explicitly accounted for the possible joint effects of system complexity and task execution in their design [14-19, 21].

---

\(^1\) Independent team programming has also been referred to simply as “team programming” by some researchers [14, 22]. While our definition is consistent with prior studies, we will use the term “independent programming” to differentiate it sufficiently from the alternative “collaborative programming” task strategy which also uses teams.
Our study of a maintenance task done by pair of programmers specifically focuses on the differences between two different team task strategies – independent programming and collaborative programming, and how the interaction between the structural software properties and the team task strategies influences maintenance performance. The central research question answered by this study is this: *What is the effect of task 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 and ease-of-use perceptions for the two different task 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 as much as 50.2% less effort than programmers employing the collaborative task strategy. But, in environments with higher structural complexity levels, teams using the collaborative programming task strategy required up to 49.5% less effort than teams using the independent programming task strategy in the same setting. Further, programmers’ perceptions of ease-of-use for modules with high cohesion and low coupling were approximately 30% higher than more complex modules, and all else being equal, the collaborative programming task strategy was perceived to be easier to use (approximately 28% higher) than the independent programming strategy. The results of this study highlight how the programmer task strategies and the structural design properties of a system can interact to jointly influence managerial variables such as maintenance effort and ease-of-use perceptions.

---

2 Although see Arisholm et al. [46] for a recent notable exception which uses control delegation (centralized vs. 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.
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 concludes the paper by highlighting the contributions of this study.

2. Theoretical Background

Two main theoretical perspectives were utilized for hypothesis development. The properties of software maintenance tasks undertaken by collaborating team members were conceptualized using the distributed cognition theoretical framework [26]. The impact of the task strategy employed by the team and its impacts on maintenance performance were analyzed using the task interdependence theoretical framework [23, 25, 27].

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 [28, 29]. 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 [26, 30]. Flor and Hutchins [22] were among the pioneers in the application of the distributed cognition framework to the study of software maintenance activities. Rogers and Ellis [31] 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 [32]. 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:
a) structure and frequency of tasks, b) team structure and the coordination mechanisms used, c) tools, documents, and the patterns of use of these artifacts, and d) development of shared knowledge. These properties derived from the distributed cognition framework are utilized for the design of this study. A pair of programmers and the software application they worked on was treated as a distributed cognition system. The activities of the programmer pairs and their task strategies (work division and coordination mechanisms) were observed, and the structural properties of the software system they worked on were controlled.

2.2. Task Interdependence Framework and Team Task Strategies

Task interdependence is the degree to which team members must rely on each other to perform their tasks [23, 25, 27]. Prior research in organizational studies and psychology have 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 [23, 25]. 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 [25, 27]. In contrast, under team interdependence, team members jointly diagnose, problem solve, 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 maintainers had the opportunity to work in parallel and independently, and this group, referred to as the independent programming task strategy, corresponds most closely to the theoretical framework.

3 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 [25, 27]. These theoretical forms of interdependence are not applicable in our research context, and thus are not considered further here.
“reciprocal interdependence”. Another group (pair of maintainers) worked jointly to complete a maintenance task, and is referred to as the collaborative programming task strategy; this group corresponds most closely to the theoretical “team interdependence” structure.

By mapping the programming task 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 task interdependence of team members can significantly impact group performance and the perceived effectiveness of team members and that selection of appropriate task strategies often requires a careful assessment of the information processing and coordination demands of team tasks [24, 25, 33, 34]. In this study we use the structural complexity of software to examine the information processing and coordination demands placed on team members working on a maintenance task, and assess the congruency between the structural complexity of software and the task strategy employed by team members for analyzing maintenance performance.

2.3 Structural Complexity of Software

There is a rich body of software engineering literature associating the structural properties of systems with their maintainability [2, 4-6, 35-37]. 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 is a measure of the interdependencies between software elements in a system, and cohesion is a measure of 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 [38-43]. Automated tools to gather coupling and cohesion metrics from
existing software systems are commercially available [44]. 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, 45]. 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 design decisions [5].

2.4. Maintenance Performance

Similar to prior software maintenance studies [5, 11, 46], performance of a team was assessed by measuring maintenance effort operationalized as the total person minutes spent by a programming team to complete a perfective maintenance task. In addition to maintenance effort programmers’ perceptions of the ease-of-use of conducting maintenance activities using the chosen task strategy were also captured. People’s subjective beliefs on ease-of-use have been shown to act as influential behavioral determinants of accepting technology and processes irrespective of their inherent objective qualities [47-49]. Therefore, it can be helpful to assess programmers’ perceptions of ease-of-use of maintenance task strategies, along with other objective measures of performance, such as maintenance effort, to assess the relevance of different task strategies while planning resource allocation policies and project work breakdown structures. Similar to past studies, perceived ease-of-use is defined as a programmer’s subjective appraisal (on Likert scales of 1-5) of the ease of conducting the maintenance tasks that were assigned to them [47-49].

3. Hypotheses

3.1. Coupling, Cohesion and Maintenance Effort

The main effects of coupling and cohesion on maintenance effort are well documented, and our first set of hypotheses is designed to check those results in our 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. On the other hand, 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 between coupling and cohesion impact maintenance effort may exist. 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.

3.2. Task Strategy and Maintenance Effort

Two specific task strategies for teamwork, independent programming and collaborative programming, are considered in this study. Under the independent programming task 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 [20, 50]. Moreover, additional effort has to be spent on achieving common ground when team members deal with errors at the boundary of their individual work.
On the other hand, under the collaborative programming task 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 task strategy. Thus, the final relative performance of independent programming and collaborative programming with regards 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. It is expected that, in such regimes, effort savings arising from parallel work enabled by the independent programming task strategy would outweigh the costs of overhead efforts (coordination and boundary spanning) associated with it. Such savings are not possible under the collaborative programming strategy because it does not facilitate parallel work. In contrast, in high coupling/low cohesion environments 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).

Thus, levels of cohesion, coupling and their interactions with the chosen task strategy are expected to significantly determine maintenance effort spent on a task. This leads to our second set of hypotheses:

**H4:** For the more highly cohesive programs, the independent programming task strategy will be associated with lower relative maintenance effort.

**H5:** For the more highly coupled programs, the collaborative programming task strategy will be associated with lower relative maintenance effort.

**H6:** Under the collaborative programming task strategy for the more highly coupled programs, maintenance effort will be lower if cohesion levels are high.

**H7:** Under the independent programming task strategy for the more highly cohesive programs, maintenance effort will be higher if coupling levels are high.
3.3. Task strategy and Perceived Ease-of-Use

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 [51-53]. Initial anchoring might be based on an individual’s prior knowledge and inherent beliefs, and adjustment to the 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 perceived ease-of-use [51]. 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-use of system maintenance.

It is expected that the collaborative programming task 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 task 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. And following the logic of hypothesis 2, it is also likely that higher levels of coupling act as an additional dampening mechanism for easy interaction between collaborating programmers, which further hinders the formation of positive ease-of-use perceptions. Therefore, our next set of hypotheses is:
H8: The perceived ease-of-use of the collaborative programming task strategy will be higher than the perceived ease-of-use of the independent programming task strategy.

H9: The difference in perceived ease-of-use 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 Experimental Procedures

4.1. Experiment Design

A controlled lab experiment method was chosen to collect data for testing the hypotheses. A 2 X 2 X 2 between-subjects experiment design, as shown in Figure 1, was chosen with the following three factors: a) coupling (low-high), b) cohesion (low-high), and c) task strategy (independent programming and collaborative programming) which generates 8 (2^3) possible conditions.

![Figure 1. Experiment Design](image)

The dependent variables were maintenance effort and perceived ease-of-use. As mentioned above, maintenance effort was measured in person-minutes, and perceived ease-of-use as the average score of a three-item questionnaire with responses sought on 5-point Likert scales. Coupling and cohesion were measured using two of the CK object oriented metrics (coupling using CBO; cohesion using LCOM) [41].

---

4. The questionnaire sought responses from the team members on (1) ease of understanding the business logic of the system while working 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-use of performing the maintenance task.
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 experiment artifact. An extensive manual inspection of the source code was conducted, along with an object-oriented metrics extraction using the CKJM tool [54]. Four different versions of the system with the same functionality, but with varying levels of coupling and cohesion corresponding to the experiment design, were developed from the original application. Coupling was primarily manipulated by modifying method calls, and cohesion was primarily manipulated by adjusting the sharing of instance variables between method pairs. The manipulations yielded two distinct levels of cohesion (low, LCOM=45; high, LCOM=10) and coupling (low, CBO=6; high, 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%). Two independent experts (not the authors) verified the functional similarity, quality, and metrics collection accuracy of the four versions of the system. A perfective maintenance task was designed to be completed by all subjects (pairs of professional programmers). The context of the perfective maintenance task was that the organization had instituted a new location, and a subset of its operational activities was to be run at the new location. Subjects were asked to modify the application in order to accommodate the new user requirements (the model solution would entail modifications of four specific classes of the application). Test data and sample reports were provided to all subjects.

---

5 Verification was done using a two stage process. During the first stage the experts reported seven errors after inspecting and testing the code. Five errors were related to variance in formatting comments and indentation schemes. Two errors were related to misplaced prefix operators instead of postfix operators in a modified version of the system. All errors reported after the first stage verification were rectified and checked in the second stage of verification, which revealed no errors in the experiment system set-ups.
4.3. Pretest and Power Analysis

The experimental system and planned procedures were pre-tested using two pairs of professional programmers and four pairs of advanced university students majoring in Information Systems. The data collected from the pretests were used to conduct a power analysis to estimate the sample size required for the experiment design. Similar to past software engineering research, the desired power for the model was chosen as 0.8, the effect size was based on the task completion rate, cell means, and standard deviations from the pretest data, and the alpha was set to 0.05 [46]. The power analysis indicated that approximately forty pairs, or eighty programmers, were needed for the 2 X 2 X 2 fixed effect 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 two years of Java programming language experience and possessing an official “Java Programmer” certification [55]. Volunteer programmers were solicited through a professional special interest group on Java programming in Singapore, the site of the experiment. Email 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

---

6 Note that in the actual experiment only professional programmers were used. In the pre-test some advanced university students were used in order to save some of the relatively more limited professional programming resources for the actual experiment.
from the main experiment tasks, but were designed to help the subjects become familiarized with the different elements 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 task 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 solution 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 post experiment survey questionnaire and were compensated 25 SGD\(^7\) for their participation in the experiment. All subjects completed the experiment within the planned two hours, and there were no dropouts.

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 non-intrusive 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 experiment data from observer notes, screen capture videos, and the version control system served to minimize any measurement related human errors.

\(^7\) SGD = Singapore Dollar. At the time of the experiment one SGD was worth approximately 0.73 US Dollars.
5. Analysis and Results

5.1. Data Analysis

Data from the experiment were analyzed using version 11 of the STATA statistical package [56]. In the first stage the normal distribution of the response variables, maintenance effort and perceived ease-of-use, were verified using the Shapiro-Wilk test [57] and through visual inspection of QQ plots, skewness and kurtosis graphs [58]. 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. All data on the two candidate cases were thoroughly checked and no errors were found, and therefore these cases were retained in the dataset (the final results are robust to both the inclusion and the omission of the two candidate cases).

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

<table>
<thead>
<tr>
<th>Covariate</th>
<th>Units</th>
<th>Min</th>
<th>Max</th>
<th>Mean</th>
<th>Std. Dev</th>
</tr>
</thead>
<tbody>
<tr>
<td>Age</td>
<td>Years</td>
<td>21</td>
<td>26</td>
<td>23.88</td>
<td>1.70</td>
</tr>
<tr>
<td>Java Experience</td>
<td>Years</td>
<td>2</td>
<td>4</td>
<td>2.40</td>
<td>0.58</td>
</tr>
<tr>
<td>Programming Career Experience</td>
<td>Years</td>
<td>2</td>
<td>8</td>
<td>3.72</td>
<td>1.55</td>
</tr>
<tr>
<td>No. of programming languages</td>
<td>Absolute number</td>
<td>2</td>
<td>5</td>
<td>3</td>
<td>1.17</td>
</tr>
<tr>
<td>Undergraduate GPA</td>
<td>Absolute number</td>
<td>3.19</td>
<td>3.9</td>
<td>3.46</td>
<td>0.30</td>
</tr>
</tbody>
</table>
Since the research model of this study involved two response variables (maintenance effort and perceived ease-of-use), a Multivariate Analysis of Variance (MANOVA) was performed, and Table 3 shows the results of this analysis. The overall model was statistically significant (Table 3, Row 1, $F=2.95$, $P$-value=0), confirming that there were significant differences in the means of maintenance effort and perceived ease-of-use 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 usual levels. The two-way interaction between coupling and cohesion (Table 3, Row 4) is also significant in the overall model at the $\alpha = .10$ level. The main effect of task strategy was not significant at usual levels (Table 3, Row 5), but the two-way and three-way interactions of task strategy with coupling and cohesion were found to be statistically significant (Table 3, Rows 4-8). Separate univariate Analysis of Variance (ANOVA) analyses were also performed for both maintenance effort and perceived ease-of-use. The univariate models were statistically significant, and results of these univariate models were similar to the MANOVA analysis (maintenance effort model: $F=5$, $P$-value=0.001, adj. $R$-squared=0.39; perceived ease-of-use model: $F=2.9$, $P$-value=0.01, adj. $R$-squared=0.23).

### Table 2. Correlations for Maintenance Effort and Perceived Ease-of-use with Potential Covariates

<table>
<thead>
<tr>
<th>Variables</th>
<th>Age</th>
<th>Java Experience</th>
<th>Programming Career Experience</th>
<th>No. of programming languages</th>
<th>Undergraduate GPA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maintenance Effort</td>
<td>0.14</td>
<td>-0.04</td>
<td>-0.03</td>
<td>0.13</td>
<td>-0.09</td>
</tr>
<tr>
<td></td>
<td>(0.30)</td>
<td>(0.79)</td>
<td>(0.86)</td>
<td>(0.38)</td>
<td>(0.58)</td>
</tr>
<tr>
<td>Perceived Ease-of-use</td>
<td>0.24</td>
<td>0.09</td>
<td>-0.13</td>
<td>-0.08</td>
<td>0.11</td>
</tr>
<tr>
<td></td>
<td>(0.11)</td>
<td>(0.54)</td>
<td>(0.41)</td>
<td>(0.6)</td>
<td>(0.46)</td>
</tr>
</tbody>
</table>

*Note: P-Values in parenthesis

### Table 3. MANOVA Analysis for Maintenance Effort and Perceived Ease-of-use

<table>
<thead>
<tr>
<th></th>
<th>Pillai’s trace statistic*</th>
<th>F-statistic</th>
<th>P-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 MANOVA Model (Adj. $R^2=0.39$; n=45)</td>
<td>1.03</td>
<td>2.95</td>
<td>0.00</td>
</tr>
<tr>
<td>2 Cohesion</td>
<td>0.28</td>
<td>4.57</td>
<td>0.01</td>
</tr>
<tr>
<td>3 Coupling</td>
<td>0.24</td>
<td>3.78</td>
<td>0.02</td>
</tr>
<tr>
<td>4 Cohesion*Coupling</td>
<td>0.18</td>
<td>2.61</td>
<td>0.07</td>
</tr>
<tr>
<td>5 Task Strategy</td>
<td>0.04</td>
<td>0.5</td>
<td>0.68</td>
</tr>
<tr>
<td>6 Cohesion*Task Strategy</td>
<td>0.19</td>
<td>2.72</td>
<td>0.06</td>
</tr>
<tr>
<td>7 Coupling*Task Strategy</td>
<td>0.17</td>
<td>2.36</td>
<td>0.09</td>
</tr>
<tr>
<td>8 Cohesion<em>Coupling</em>Task Strategy</td>
<td>0.21</td>
<td>3.08</td>
<td>0.04</td>
</tr>
</tbody>
</table>

\*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 [59].
5.2. Hypothesis Tests

The individual hypotheses developed in section 2 were examined using post-hoc tests. 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 [60], and these results are shown in Table 4.

<table>
<thead>
<tr>
<th>No.</th>
<th>Hypothesis</th>
<th>Observed Difference</th>
<th>Statistically Supported?</th>
<th>Statistical Test Result†</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>H1</td>
<td>Maintenance effort is lower for the more highly cohesive programs</td>
<td>High Cohesion 47% less effort than Low Cohesion</td>
<td>Yes</td>
</tr>
<tr>
<td>2</td>
<td>H2</td>
<td>Maintenance effort is higher for the more highly coupled programs</td>
<td>High Coupling 80% higher effort than Low Coupling</td>
<td>Yes</td>
</tr>
<tr>
<td>3</td>
<td>H3</td>
<td>For the more highly coupled programs, maintenance effort is lower if cohesion levels are high</td>
<td>[High Coupling/Low Cohesion] 56% less effort than [High Coupling/Low Cohesion]</td>
<td>Yes</td>
</tr>
<tr>
<td>4</td>
<td>H4</td>
<td>For the more highly cohesive programs, independent programming task strategy is associated with lower maintenance effort.</td>
<td>Independent programming 11% less effort than Collaborative programming</td>
<td>No</td>
</tr>
<tr>
<td>5</td>
<td>H5</td>
<td>For the more highly coupled programs, collaborative programming task strategy is associated with lower maintenance effort.</td>
<td>Collaborative programming 12% less effort than Independent programming</td>
<td>No</td>
</tr>
<tr>
<td>6</td>
<td>H6</td>
<td>Under the collaborative programming task strategy, for the more highly coupled programs, maintenance effort is lower if cohesion levels are high</td>
<td>[High Coupling/Low Cohesion] 63% less effort than [High Coupling/Low Cohesion]</td>
<td>Yes</td>
</tr>
<tr>
<td>7</td>
<td>H7</td>
<td>Under the independent programming task strategy, for the more highly cohesive programs, maintenance effort is higher if coupling levels are high</td>
<td>[High Cohesion, High Coupling] 122% higher effort than [High Cohesion/Low Coupling]</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Perceived Ease-of-use Hypotheses

<table>
<thead>
<tr>
<th>No.</th>
<th>Hypothesis</th>
<th>Observed Difference</th>
<th>Statistically Supported?</th>
<th>Statistical Test Result†</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>H8</td>
<td>Perceived ease-of-use of the collaborative programming task strategy will be higher than perceived ease-of-use of the independent programming task strategy</td>
<td>Collaborative programming 28% higher ease of use than Independent programming</td>
<td>Yes</td>
</tr>
<tr>
<td>9</td>
<td>H9</td>
<td>The difference in perceived ease-of-use between collaborative programming task strategy and independent programming task strategy decreases with an increase in coupling levels of programs</td>
<td>[Collaborative programming – Independent programming] for High Coupling is 86% lower ease of use than [Collaborative programming – Independent programming] for Low Coupling</td>
<td>Yes</td>
</tr>
</tbody>
</table>

† All P-values are two-tailed; Bonferroni adjusted P-values are 0.006 for 5% significance (marked as *** 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 reported in Figure 2 (maintenance effort) and Figure 4 (perceived ease-of-use). 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 $\frac{\text{high cohesion} - \text{low cohesion}}{\text{low cohesion}}$ as a percentage (i.e., $\frac{(102.2 - 192.7)}{192.7} \times 100 = -46.96$), shows that the highly cohesive programs requires about 47% 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. However, although we found significant interaction effects for task strategy in the MANOVA analysis (Table 3, Rows 4-8), a comparison of means as posited by hypotheses H4 and H5 did not reveal statistically significant results at the Bonferroni adjusted $\alpha = .006$ level. Even though the observed differences between High Cohesion and Low Cohesion groups under the independent programming strategy (refer 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 Bonferroni adjusted $\alpha = .006$ level. However, hypotheses H6 and H7, which proposed a three-way interaction between coupling, cohesion and task strategy, were strongly supported (p-value=0.000, refer Table 4 H6, H7).
Figure 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 task strategy). Figure 2 also includes the sub-division of overall maintenance effort into program comprehension, explicit coordination, and execution portions. These sub-divisions 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\(^8\).

![Figure 2. Effects of Coupling, Cohesion, and Task Strategy on Maintenance Effort](image)

The three-way interaction effects between coupling, cohesion, and task strategy on maintenance effort are visually represented in Figure 3.

\(^8\) Note that the amount of time in the fourth graph spent on “Coordination” in the High Cohesion & High Coupling, Collaborative Programming stacked histogram is so small that it does not display at normal printing resolutions.
Both of the perceived ease-of-use hypotheses (refer Table 4 H8, H9) were fully supported at the Bonferroni adjusted $\alpha=.006$ level. As we had expected the perceived ease-of-use level for the collaborative programming task strategy groups was higher than for the independent programming task strategy groups. And as hypothesized in H9 under the collaborative programming strategy highly coupled programs showed lower perceived ease-of-use levels than programs with lower levels of coupling.

The perceived ease-of-use levels for the various experimental groups are presented in Figure 4. The significant three-way interaction between coupling, cohesion, and task strategy is depicted visually in Figure 5.
Figure 4. Effects of Coupling, Cohesion, and Task Strategy on Perceived Ease-of-use

Figure 5. Interaction Effects of Coupling, Cohesion, and Task Strategy on Perceived Ease-of-use

6. Discussion and Conclusion
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 task strategies employed by maintenance teams. Supported by the theoretical perspectives of the distributed cognition and task
interdependence frameworks, this study experimentally validated that the task strategy employed by maintenance teams, along with structural complexity, are important factors in influencing variables of managerial interest such as maintenance effort and perceived ease-of-use.

Specific differences in maintenance effort across different levels of structural complexity and task strategies can be inferred from Figure 2, which shows the cell means of all the combinations of the interacting variables. Since the maintenance task across the experiment cells remained constant, the observed differences in maintenance effort can be interpreted as productivity differences (i.e., the numerator in the equivalent productivity equation (output/input = maintenance task size/effort) remained the same across the cells).

Referring to Figure 2 it can be seen that, other than in the high cohesion/low coupling quadrant (lowest structural complexity), teams using the collaborative programming task 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 task programming strategies can be seen in the low cohesion/low coupling quadrant (49.5%), and the smallest difference is found in the low cohesion/high coupling quadrant (highest structural complexity) (14%). 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% more productive on average than the programmers employing the collaborative task strategy. Irrespective of the task strategies employed, maintenance of high cohesion programs was 47% more productive than maintenance of low cohesion programs. Similarly, it required on average 80% more effort from programmers to finish maintenance tasks in highly coupled programs as compared to programs with lower levels of coupling.
Referring to Figure 4 it can be seen that programmers’ ease-of-use perceptions on the task strategies were also highly contingent on the structural complexity levels they encountered. Programmers’ perception of ease-of-use for modules with high cohesion and low coupling were 30% higher than other more complex modules, and all else being equal, the collaborative programming task strategy was perceived to be easier to use (28% higher) than the independent programming strategy. However, the ease-of-use perception difference between collaborative programming and independent programming dropped significantly (86%) as coupling increased.

These results provide evidence for the proposition that managers can 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 statically predetermined by the structure of the software alone, but are contingent on the team task strategies that are employed during maintenance. Referring to Figures 2 and 4, one can see how the task strategy contingency of structural complexity plays out, resulting in different maintenance effort levels and perceived ease-of-use for different groups of the interaction.

A variety of descriptive differences in lower order factors of program comprehension, coordination, and solution execution effort can also be noticed due to the dynamic interactions between software structure and team task strategy. 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 task 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. The contingency view of structural complexity has several implications for both research and practice, both of which are discussed below.
6.1. Primary implications for research

Software engineering research studies that compare different processes or techniques (e.g., individual vs. 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. It is posited 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 theoretical rationale using the distributed cognition and task interdependence frameworks to support a contingency view of software complexity, and experimentally validated the use of task strategy-coupling-cohesion interactions as a way to account for complexity. Task strategy has the benefits 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.

6.2. Primary implications for practice

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, such as improved productivity, that could be reaped if careful attention is paid to achieve congruence between project task work breakdown structures and team task strategies. Work breakdown structures capture the planned division of labor mechanism by managers. Task strategies, on the other hand, determine how the elements of a work breakdown structure are further chunked, clustered, or executed as-is during actual task execution. Hence, incongruence between the planned work breakdown structures and actual task strategies may result in unexpected overhead costs, such as unplanned coordination and idle time\(^9\).

---

\(^9\) Of course, achieving congruence between work breakdown structures and team strategies in practice could be challenging because of the complex optimization that may be necessary to match, among other things, resource availability.
Our study shows 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% over managers who use a simple uniform resource allocation policy. It is important to note, however, that choice of the maintenance team task strategy might be affected by programmers’ willingness to employ it, and managers should be aware that programmers may prefer a task 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 the lack of implementation of automated tools. One way to break such a deadlock is through local tailoring of processes, and through treating task 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 result in significant savings in total effort expended.
6.3. Limitations and Future Research

This research study is based on a controlled experiment. 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 (by focusing only on pairs of certified professional programmers with at least two years of experience) helped in maintaining control of the primary factors of interest, such designs necessitate trade-offs with other potential research questions. For example, this experiment leaves to future research possible manipulation of programmer expertise (novices vs. 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 dynamic structural complexity on other potentially relevant response variables, such as reuse or conformance quality.

Now that the current study has shown the effect of two consistent team task 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 task strategy, coupling, and cohesion could be examined in future research as well.

6.4. 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 task strategies of the actors (independent programming vs. 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 task strategies. Maintenance effort and perceived ease-of-use of programmers are significantly influenced by the complex three-way interactions between coupling, cohesion and task strategy. This study provides an empirically validated theoretical rationale for using the coupling-cohesion-task 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.

References


29


