MIS Quarterly

RESEARCH ARTICLE

THE ROLE OF COGNITIVE FIT IN THE RELATIONSHIP BETWEEN SOFTWARE COMPREHENSION AND MODIFICATION[1]

By: **Teresa M. Shaft**
Michael F. Price College of Business
University of Oklahoma
307 West Brooks, Room 306B
Norman, OK 73019-7482
U.S.A.
tshaft@ou.edu

Iris Vessey
University of Queensland and
Queensland University of Technology
126 Margaret Street
Brisbane Q4000
AUSTRALIA
i.vessey@qut.edu.au

Abstract

Although there is a long tradition of empirical studies of software developers, few studies have focused on software maintenance. Prior work is predicated on the belief that higher levels of software comprehension are associated with higher levels of performance on modification tasks. This study provides a more complete understanding of the relationship between software comprehension and modification. We conceptualize software maintenance as interlinking comprehension and modification, and argue that the relationship between the two is moderated by cognitive fit. Specifically, cognitive fit exists when the software maintainer's dominant mental representation of the software and their mental representation of the modification task emphasize the same type of knowledge. We hypothesize that when cognitive fit exists, greater improvements in comprehension are associated with higher levels of performance on a modification task. When cognitive fit does not exist, however, the software maintainer's mental representations of the software and of the modification task do not emphasize the same type of knowledge, which may mean that attention is devoted to comprehension at the expense of modification, resulting in lower performance on the modification task. In these circumstances, comprehension and modification tasks may interfere with each other, an effect known as dual-task interference. We therefore hypothesize that performance on a modification task is moderated by the fit between the mental representation of the software and that of the modification task.

We tested our theory by varying cognitive fit to create matched and mismatched conditions in a single experiment that used IT professionals as subjects. Our findings support our theory: cognitive fit moderates the relationship between comprehension and modification. Specifically, changes in software comprehension and modification performance are positively related when cognitive fit exists and negatively related when cognitive fit does not exist. Our findings demonstrate the need to examine more complex relationships among the numerous types of tasks involved in software development rather than examining software comprehension alone.

Keywords: Theory of cognitive fit, software modification, software comprehension, dual-task interference

Introduction

Software maintenance is a high-cost, low-productivity activity (Sharon 1996) that accounts for 80 percent of the effort associated

[1]Peter Todd was the accepting senior editor for this paper.

with the software function (Hatton 1998). Despite its importance, maintenance is an understudied phenomenon (Kemerer and Slaughter 1999). Maintenance studies frequently focus on characteristics of the software such as program complexity, project size, and added functionality to assess how these characteristics are related to later maintenance efforts (see Banker et al. 1998). Such descriptive studies typically assess an organization's maintenance portfolio *post hoc* with the aim of providing advice on how to manage these characteristics to improve the quality or maintainability of the software. Such software-level approaches do not, however, address the fact that software maintenance is conducted by an IT professional who must change the software to meet the new requirements.

The human element in software work has long been known to be problematic; for example, in 1968, Sackman et al. conducted debugging studies that revealed a 1:26 variability in time to debug software. Despite the early establishment of a tradition of studying software developers, relatively few studies have examined performance on software maintenance tasks. Prior maintenance research has considered the impact of various factors on maintainers' ability to modify software including program structure (Boehm-Davis et al. 1992; Daly et al. 1996; Prechelt et al. 2001; Sheppard et al. 1979; Shneiderman et al. 1977), the use of different documentation formats (Curtis et al. 1989; Prechelt et al. 2002), and the cognitive processes used by software maintainers (Littman et al. 1986).

Rather than focusing on software maintenance *per se*, researchers have more often studied software comprehension. Software comprehension involves understanding existing software that frequently is written by another developer (Lientz et al. 1978), while modification involves changing, deleting from, or adding to the software. Comprehension studies have sought to understand software developers' mental representations (Corritore and Weidenbeck 1999; Pennington 1987b), the cognitive processes they use during comprehension (Koenemann and Robertson 1991; Shaft and Vessey 1995; Vans et al. 1999; von Mayrhauser and Vans 1995; von Mayrhauser et al. 1997), and how characteristics of the software influence comprehension (Shaft and Vessey 1998).

Typically, if a developer modifies software in a *comprehension* study, the modification task is a way of engaging the developer in studying the software; comprehension is assessed before and after conducting the modification, but performance on the modification task itself is not examined. One of the reasons behind the focus on software comprehension may be that it underlies a number of software-related tasks, in particular, maintenance and debugging. Robson et al. (1991), for example, estimate that 50 to 90 percent of the effort during software maintenance is spent on comprehending the software, with the remaining 10 to 50 percent of the effort devoted to changing it. Further, those researchers who have addressed modification have assumed that higher levels of comprehension lead to higher levels of performance (see, for example, Hendrix et al. 2002).

Current understanding of software maintenance, however, leads one to question a direct link between comprehension and modification performance. For example, maintenance personnel often successfully modify software without understanding it fully (see, for example, Corritore and Wiedenbeck 1999, Littman et al. 1986, Pennington 1987b). Further, given the size of many industrial applications, it seems unlikely that maintenance personnel could devote the time necessary to develop a complete understanding of the entire application in the course of conducting a modification. Hence, while comprehension is an important aspect of software development, the quest to find ways to help developers achieve high levels of comprehension to improve performance across a number of software tasks may oversimplify the relationship between comprehension and other software tasks.

Little research has addressed the relationship between comprehension and modification empirically. Further, studies that have examined both comprehension and modification per se have done so in separate experiments rather than in a single experiment that would allow researchers to determine interrelationships (see, for example, Curtis et al 1989; Sheppard et al. 1979; Shneiderman et al. 1977).

The purpose of this study is to examine the relationship between software comprehension and modification to provide a more thorough understanding of the interrelationships between the two tasks. We propose that the relationship is more complex than previously assumed and that high levels of software comprehension may not always translate into high levels of performance on modification tasks. We propose, instead, that the relationship between comprehension and modification is moderated by the cognitive fit between the knowledge emphasized in the software maintainer's mental representation of the software and the requirements of the software modification task. Because all maintenance tasks require both comprehension of the original software and making the required changes to the software (i.e., the modification), we believe that our theory may well apply to other types of maintenance tasks, such as removing features and making changes, as well as to the enhancements investigated in this research. Our focus in this research is on enhancement because it consistently represents the largest category among the various software maintenance tasks (see, for example, Barry et al. 1999; Lientz et al. 1978).

In the next section, we present theories relevant to our study of the interrelationship between software comprehension and modification, and develop our conceptual model. We then present our propositions, from which we develop testable hypotheses. Next we present the experimental methodology used to test the hypotheses, followed by the results. Further, we evaluate two theoretical alternatives to lend strength to the test of our theoretical arguments. Finally, we present the implications of our study for research and practice.

Theoretical Development

In presenting the theoretical underpinnings for our study, we use *information* to refer to the contents of the software and *knowledge* to refer to the software maintainer's mental representations of a software system. We first present the theoretical basis for our study

and then address how we applied the theory to our specific context of modifying computer programs.

Theory on Dual-Task Problem Solving

The basic premise of our theoretical model is that software maintenance involves both comprehending the software and making the modification; that is, the task as a whole can be viewed as having two readily identifiable, interrelated subtasks. Hence we develop a dual-task problem-solving model to explain how problem solving takes place when the problem solver must attend to two interrelated tasks.

Dual-Task Problem-Solving Model

We use as the starting point for developing our theoretical model the general form of the problem-solving model used by Vessey (1991) to describe the theory of cognitive fit (Figure 1). This model views problem-solving performance as resulting from the interaction between the external representation and the problem-solving task. In keeping with the traditions of cognitive science, this model views processing as taking place within the mental representation (Zhang 1997). To support the solution of the task, the problem solver develops a consistent mental representation that is based on internalization of information in the external representation and any internal representations relevant to task solution.

We elaborate on this model in two ways for the purpose of our investigation into dual-task problem solving. First, we enrich the model using the work of Zhang and Norman (1994), who distinguish the internal representation from the mental representation that is constructed to solve a problem.[2] This approach explicitly addresses both internal and external representations as well as a mental representation, in what they refer to as a distributed model of cognition (see Zhang 1997). Hence we modify Figure 1 to reflect the fact that both the *internal* and *external representations*, and the interactions among them, contribute to the *mental representation for task solution* that is developed to solve the problem. (Note that we use *italics* to highlight references to specific constructs in all of our models.) Figure 2 presents the general model of problem solving that incorporates notions of distributed cognition. As an example, the task of understanding a piece of software is influenced by both the software itself (*external problem representation*) and the maintainer's existing knowledge of the software domain (*internal representation of the problem domain*), as well as the task that is required to be completed (*problem-solving task*). Task solution is accomplished via the *mental representation for task solution*.

Second, because we conceive of software maintenance as consisting of two interrelated subtasks, we present the general model of IS problem solving on which our investigation of software modification is based as a dual-task model. Figure 3 presents a general theoretical model of interacting tasks in the context of software maintenance. The model essentially consists of three cognitive fit models, one for each subtask and one for the interaction between them. Problem solvers form mental representations for each of the component tasks. The maintainer forms a *mental representation of the software* based on general knowledge of software and software development and specific knowledge of the software at hand. The maintainer develops a *mental representation of the modification task* based on general knowledge of software development and specific knowledge of the modification task *(external representation of the modification*). While conducting the modification, the maintainer will engage in further comprehension resulting in changes to his/her *mental representation of the software*. To complete the modification, the maintainer must then integrate the two mental representations into a *mental representation for task solution*, which is manifested in *problem-solving performance*.

Formulating the issue of making software modifications as a dual-task model opens the way for us to consider situations in which one task might either facilitate or inhibit the other.

Roles of Dual-Task Interference and Cognitive Fit in Dual-Task Problem Solving

The phenomenon of dual-task interference, which occurs when problem solvers perform two (or more) tasks simultaneously, has been investigated by psychologists for several decades. It is manifested in performance degradation on one or both of the tasks that are being addressed simultaneously (see, for example, Durso et al. 1998; Navon and Gopher 1979; Pashler 1994; Wickens 2002). When dual-task interference occurs, it is difficult for the individual to allocate attention effectively between the interacting tasks (Durso and Gronlund 1999), resulting in reduced performance (Van Selst and Jolicoeur 1997).

Much of the research in the area has focused on the resources needed to conduct the two tasks simultaneously and, therefore, the allocation of resources between them (see, for example, Durso and Gronlund 1999; Kahneman 1973; Wickens 2002). Although there is still substantial debate regarding the under lying mechanisms, the effects have been observed consistently (see, among others, Navon 1990; Navon and Miller 1987; Pashler 1994; Pashler and O'Brien 1993; Sarno and Wickens 1995). Hence, we apply the basic premises of dual-task interference to our specific context of IS problem solving.

Allocating attention between tasks is particularly relevant to software maintenance because maintainers must divide their attention between comprehending the existing software and making relevant changes to the software. Comprehending the software, alone, is a substantive task because maintainers must divide their attention among multiple elements within the software itself (e.g., different modules), as well as understand information other than that

[2]Note that Zhang and Norman (1994) use the term *task space* rather than the term *mental representation* used here.

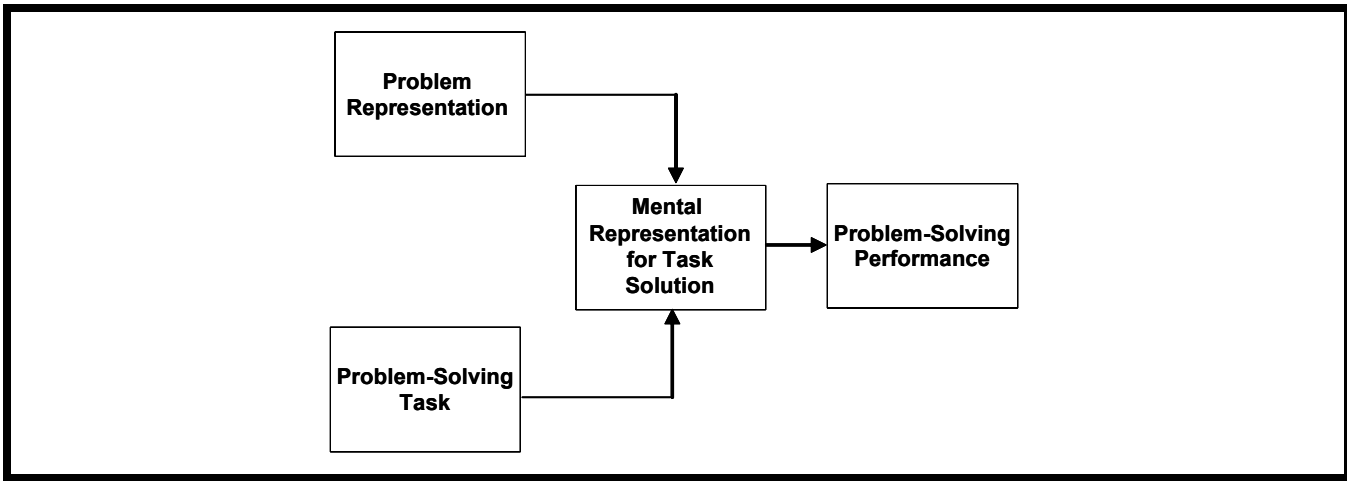

Figure 1. Cognitive Fit in Problem Solving

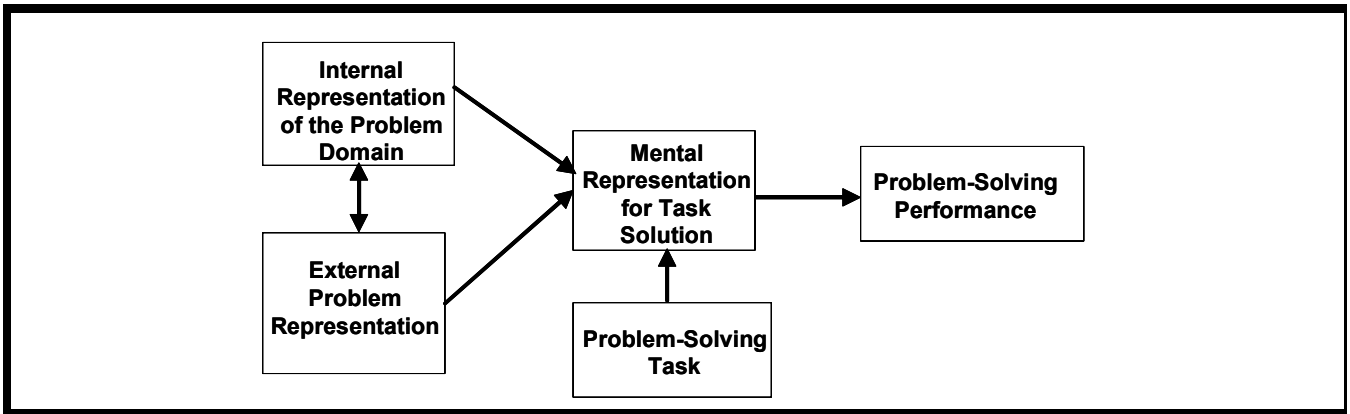

Figure 2. Extended Cognitive Fit Model

in the software (e.g., the external software documentation, etc.). Modifying software is also a substantive task as maintainers must both understand the modification to be conducted as well as make appropriate changes to the software.

We again draw on the theory of cognitive fit (Vessey 1991) to provide the theoretical basis for what happens when knowledge of two tasks is needed for problem solving. When one is solving a problem made up of two tasks, two possible types of interaction may result: the two tasks may be consistent or compatible with each other, or they may not.

Matching Representations

When cognitive fit exists, the software maintainer's *mental representation of the software* and their *mental representation of the modification task* emphasize similar types of knowledge and have similar requirements. Hence no transformations are required to form the *mental representation for task solution* and the cognitive requirements are effectively reduced (Vessey 1991), thereby increasing the maintainer's ability to allocate attention between tasks (see, also, Durso and Gronlund 1999). A maintainer is able to shift attention relatively easily, therefore, between comprehension and modification tasks with comprehension efforts being directed toward the information embedded in the software that is most relevant to the modification. As a result, *problem-solving performance* is likely to be more accurate and quicker than would otherwise be the case.

This notion is supported by a number of authors who have observed that problem solvers have a greater ability to perform two tasks that are compatible (rather than incompatible) at the same time, thus reducing the impact of dual-task interference (see, for example, Koch and Prinz 2002; Whitaker 1979). We state the following proposition:

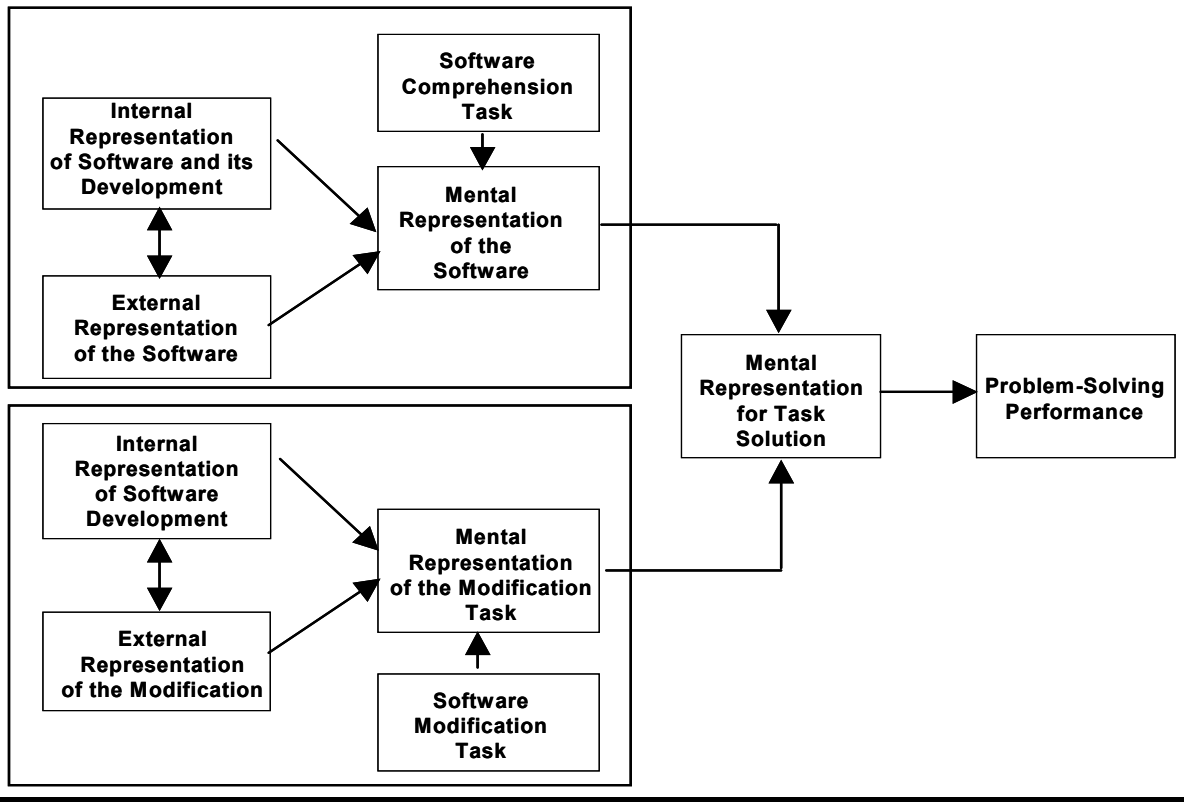

Figure 3. General Model of Interacting Tasks in Software Maintenance

> **Proposition 1:** When the *mental representation of the software* is consistent with the *mental representation of the modification task*, **increases** in software comprehension are associated with **higher** levels of performance on the modification task.

Mismatching Representations

When cognitive fit does not exist (that is, when knowledge in each of the two task areas does not mutually support problem solving), the software maintainer's *mental representation of the software* and the *mental representation of the modification task* needed to support task solution emphasize different types of knowledge. Dual-task interference occurs, and the maintainer may experience difficulty in allocating attention effectively between comprehension and modification tasks that are not mutually supportive. In other words, without cognitive fit, there is nothing to guide the maintainer in working toward task completion (Vessey 1991). As a result, comprehension activities are likely to interfere with the maintainer's ability to complete the modification task, and performance suffers.

In these circumstances, some kind of transformation must be brought about before problem solving can proceed: either the *mental representation of the software* must be transformed to emphasize similar knowledge to that in the *mental representation of the modification task*, or vice versa. Hence the maintainer confronted with these challenges may take one of two approaches to resolve the situation: (1) focus further on comprehending the software or (2) focus on performing the modification. In the first case, because the *mental representation of the software* tends to drive the comprehension process (Burkhardt et al. 2002), there is a tendency for the maintainer to heed information in the software that is consistent with the knowledge already emphasized in the *mental representation of the software*, rather than the knowledge consistent with their *mental representation of the modification task* (see Broadbent 1971). Changing a *mental representation of the software* to reflect an increased understanding of knowledge that is not consistent with their *mental representation of the modification task* does not, however, enhance the maintainer's ability to modify the software. Hence, both efficiency and effectiveness are likely to be affected and it is likely that improved levels of comprehension will be associated with lower levels of *problem-solving performance*.

On the other hand, a software maintainer might focus on the modification task rather than seeking to further comprehend the software. Because problem solvers who focus on task solution perform more effectively than those who focus more on the present

state (Durso et al. 1998; Hogg et al. 1995; Vessey 1991), the software maintainer might be better served by taking this approach, which would then require switching attention to their *mental representation of the software* only when necessary to resolve an issue directly related to the modification task. Focusing on the modification task is also difficult, however. First, because the type of knowledge emphasized in a maintainer's *mental representation of the software* tends to be quite stable over time (Corritore and Wiedenbeck 1999), maintainers have difficulty shifting to a different *mental representation of the software* after invoking an inappropriate one (Taylor et al. 1997). Second, when a maintainer attempts to acquire knowledge consistent with that required to conduct the modification task, it is difficult to map that knowledge into their mismatched *mental representation of the software*. The process of building up the knowledge essential to conducting the modification task (that is, developing the *mental representation for task solution*) is quite challenging and, again, both efficiency and effectiveness are likely to be affected. Therefore, when a maintainer focuses on the modification task, they will likely make few gains in comprehension.

Based on the arguments above, therefore, when there is a mismatch between the maintainer's *mental representation of the software* and their *mental representation of the modification task*, *problem-solving performance* will be inversely related to improvements in comprehension. Hence, we state the following proposition:

> **Proposition 2:** When the *mental representation of the software* is inconsistent with the *mental representation of the modification task*, **greater** increases in software comprehension are associated with **lower** levels of performance on the modification task, or **higher** levels of performance on the modification task are associated with **lesser** increases in software comprehension.

In summary, then, when the approach to comprehension does not support the task to be conducted, (1) increasing attention to comprehension of the software distracts the programmer from the primary task of modification or (2) focusing on the modification task interferes with the programmer's understanding of the software. In both cases, the relationship between comprehension and modification performance is an inverse one.

Theory on Mental Representations in the Domain of Software

We now apply the notions of dual-task problem solving to our specific task of modifying computer programs by examining mental representations in the software domain in order to determine what constitute matching and non-matching situations in this domain. We first examine the intrinsic characteristics of software. We then apply that knowledge to the *mental representations of software* that software maintainers form, as well as to the types of modification tasks (*software modification task*) that they may be requested to accomplish, which is reflected in their *mental representation of the modification task.*

We address the characteristics of software and software tasks via the many types of information that are embedded in a piece of software. Numerous researchers have addressed the issue in terms of the types of information embedded in a computer program (see, for example, Brooks 1987; Curtis et al. 1989; Green 1977; Pennington 1987a, 1987b; Shaft and Vessey 1998). Part of the essential difficulty of building and maintaining software comes from the difficulty of representing these different types of information (see Brooks 1987).

Pennington's (1987a, 1987b) characterization of the types of information in software as function, data flow, control flow, and state information best encompasses the relevant research and has been used widely in studies addressing the types of information found in software. Hence we use Pennington's characterization in our research. Function information reflects the main goals of the program and the hierarchy of subgoals. Data flow information reflects the series of transformations that data objects undergo. State information relates to the condition-action information embedded in a program (i.e., the program actions that result when a set of conditions is true). Control flow information reflects execution sequence (i.e., the order in which actions occur).

Mental Representations of Software

Software maintainers incorporate these different types of information into their *mental representations of the software*. They may develop multiple mental representations, each of which emphasizes a particular type of information. The types of representations that they form depend on their experience with software and with their knowledge of the application domain of the software, among other factors. A series of protocol analysis studies have led to the characterization of these representations as domain, program, and situation models (Vans et al. 1999; von Mayrhauser and Vans 1995, 1996; von Mayrhauser et al. 1997).

A software maintainer's domain model focuses on software functionality (Vans et al. 1999). Therefore, the domain model emphasizes the function information contained in the software. It is a high-level model that is more closely aligned to the application domain (that is, the problem rather than the software) and is less detailed than the other two models (Vans et al. 1999; von Mayrhauser and Vans 1996).

A software maintainer's program model emphasizes how the software accomplishes tasks (von Mayrhauser and Vans 1995). It is closely aligned with the programming domain and is the most detailed of the models (Pennington 1987b). Therefore, it emphasizes the maintainer's understanding of the control flow and state information embedded in the software. Control flow information reflects the sequencing of actions within the software, while state information reflects connections between the execution of an action

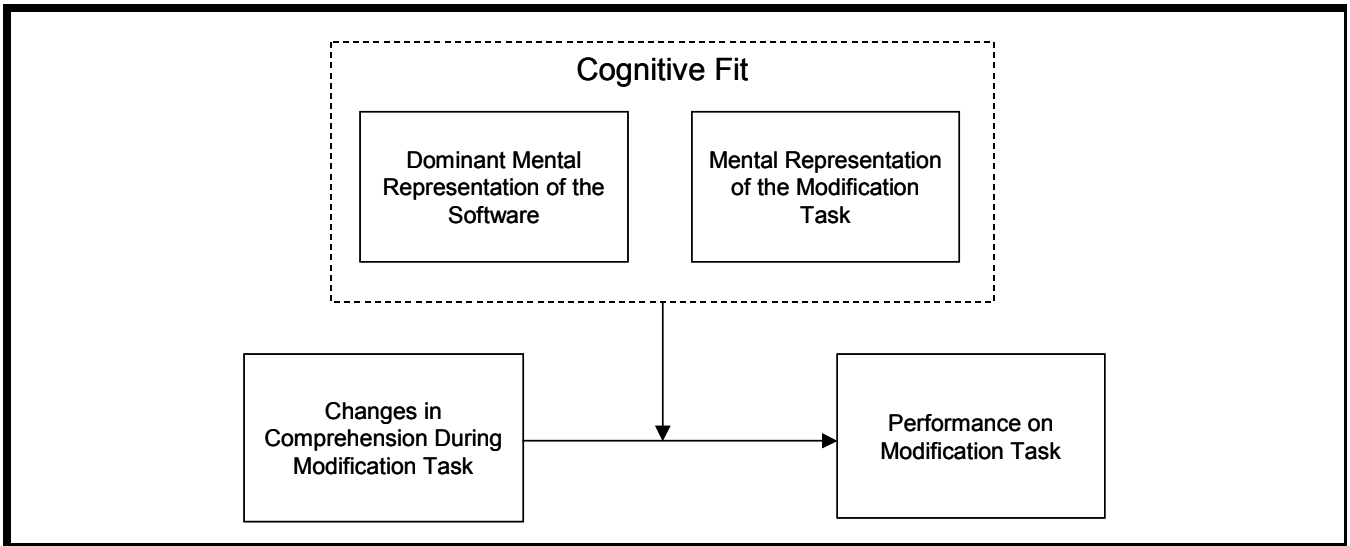

Figure 4. Research Model of the Relationships Among Cognitive Fit, Comprehension, and Modification Performance

and the state of the software when the action occurs (Pennington 1987b).

A software maintainer's situation model serves as a bridge between the domain and program models and may be characterized as representing the algorithmic level of the software (Vans et al. 1999). It allows a maintainer to avoid cognitive overload by reorganizing the knowledge gained through the line-by-line study of the software into higher-level chunks and reflects the maintainer's understanding of data flow information in the software. We can consider the domain and program models as being at the opposite ends of a continuum, with the situation model residing between them.

When engaging in software comprehension, a software maintainer invokes one of the possible *mental representations of the software* (domain, program, or situation model) (Vans et al. 1999), which then drives the comprehension process (Burkhardt et al. 2002). For ease of explication, we refer to this representation as the maintainer's dominant *mental representation of the software*, or more simply, the dominant *mental representation*. Because the domain model is linked to the application domain, software maintainers tend to invoke it when they are familiar with the application domain (von Mayrhauser and Vans 1995); that is, a maintainer's domain model will be activated when the application domain is familiar. When maintainers lack application domain knowledge, they tend to invoke the program model, thereby relying on their understanding of the programming language and standard programming constructs (von Mayrhauser and Vans 1995); that is, a maintainer's program model will be activated when the application domain is unfamiliar. The situation model is unlikely to be invoked at the outset of comprehension as it tends to develop after the program model and only after extensive interaction with a specific piece of software (von Mayrhauser et al. 1997).

Mental Representations of Modification Tasks

Software modification tasks can be conceptualized as affecting one of the types of information embedded within software more than the other types of information. We can determine the most important types of modification tasks in practice by examining prior modification studies and also the tools that embed techniques for aiding software maintenance. Prior researchers, for example, have largely investigated control flow tasks (Barry et al. 1999; Curtis et al. 1989). Software metrics tools typically focus on data flow as well as control flow (Zuse 1991) and program slicing techniques extract the data flow for a particular (set of) variable(s) relevant to a modification or debugging task (Weiser 1982). In this study, we investigated a control flow modification task, which should be consistent with a program model because it emphasizes how software accomplishes tasks (von Mayrhauser and Vans 1995). We also investigated a function modification task, which should be consistent with a domain model because it emphasizes function information (Vans et al. 1999).

We conceptualize software maintainers as creating a *mental representation of the modification task* based on the requirements of the modification task (the *software modification task*), the way in which the modification task is presented (*external representation of the modification*), and their existing knowledge of software development (*internal representation of software development*). The *external representation of the modification* is a specification of the software modification task such as a narrative or graphic. Essentially, then, their *mental representation of the modification task* will emphasize the type of knowledge that is emphasized in the task requirements *(software modification task)*. The maintainer's *mental representation of the software* and *mental representation of the*

modification task may or may not match in that they may or may not emphasize the same type of knowledge.

Hypotheses

To test the propositions, we need to establish conditions for cognitive fit between the maintainer's dominant *mental representation of the software* and *the mental representation of the modification task.* We address this issue from the perspective of the dominant *mental representation* and, from there, establish match/mismatch relationships with the type of modification task. We also address the aspect of comprehension that is relevant to the research model we use to address our propositions (see Figure 4). We then derive hypotheses to test the propositions.

We manipulated the maintainer's dominant *mental representation of the software* by having them conduct modification tasks in familiar and unfamiliar application domains. We invoked the maintainer's domain model by using software from a familiar application domain (Vans et al. 1999) and their program model by using software from an unfamiliar application domain (von Mayrhauser and Vans 1995).

We operationalized cognitive fit by using two modification tasks, one of which emphasized function information, while the other emphasized control flow information. A domain model of software focuses on function information. Pennington, for example, states that functional information involves the "main goals of the program and the hierarchy of subgoals necessary to achieve this goal" (1987a, p. 101), while von Mayrhauser et al. states that "it is much easier to decompose the program into functional units if the programmer is well acquainted with the domain" (1997, p. 311). Therefore, cognitive fit between the software maintainer's dominant *mental representation of the software* (in this case, the domain model) and their *mental representation of the modification task* occurs when they perform a function-oriented modification.

In an unfamiliar application domain, the software maintainer's dominant *mental representation of the software* is the program model. As we have seen, the program model is based on control flow and state knowledge. Control flow "reflects the execution sequence of a program, the order in which actions will occur" (Pennington 1987a, p. 101). We used a control flow modification to operationalize cognitive fit when the software maintainer's dominant *mental representation of the software* is the program model.

We now address that aspect of comprehension that is relevant to our research model (see Figure 4). Our theory suggests that the fit conditions we establish will moderate the effectiveness of the comprehension that occurs during problem solving. The comprehension that occurs during problem solving is a reflection of changes to the *mental representation of the software*. The important variable in the relationship between comprehension and problem-solving performance is the change in comprehension that occurs during conduct of the modification task. We therefore motivate change in the level of comprehension observed during the modification task as the relevant aspect of comprehension in our study.

In the context of Proposition 1, we state the following hypotheses:

> **Hypothesis 1a**: When a software maintainer is familiar with the software's application domain and conducts a function-oriented modification task, **increases** in software comprehension are associated with **higher** levels of performance on the modification task.
>
> **Hypothesis 1b**: When a software maintainer is unfamiliar with the software's application domain and conducts a control flow modification task, **increases** in software comprehension are associated with **higher** levels of performance on the modification task.

Hence when the information emphasized in the programmer's *mental representation of the software* and that of the *modification task* are similar, the two tasks are mutually supportive, resulting in a direct relationship between changes in software comprehension and performance on the modification task.

Our second proposition is based on conditions in which cognitive fit does not exist. We created these conditions by invoking situations in which the knowledge emphasized in the dominant *mental representation of the software* and the *mental representation of the modification task* do not match, that is, by counter-crossing the conditions used to create cognitive fit. Cognitive fit does not exist when maintainers perform a function-oriented modification in an unfamiliar domain, and a control flow-oriented modification in a familiar domain.

In the context of Proposition 2, we state the following hypotheses:

> **Hypothesis 2a**: When a software maintainer is unfamiliar with the software's application domain and conducts a function-oriented modification task, **greater** increases in software comprehension are associated with **lower** levels of performance on the modification task, or **higher** levels of performance on the modification task are associated with **lesser** increases in software comprehension.
>
> **Hypothesis 2b**: When a software maintainer is familiar with the software's application domain and conducts a control flow modification task, **greater** increases in software comprehension are associated with **lower** levels of performance on the modification task, or **higher** levels of performance on the modification task are associated with **lesser** increases in software comprehension.

Hence when the information emphasized in the programmer's *mental representation of the software* and the *mental representation*

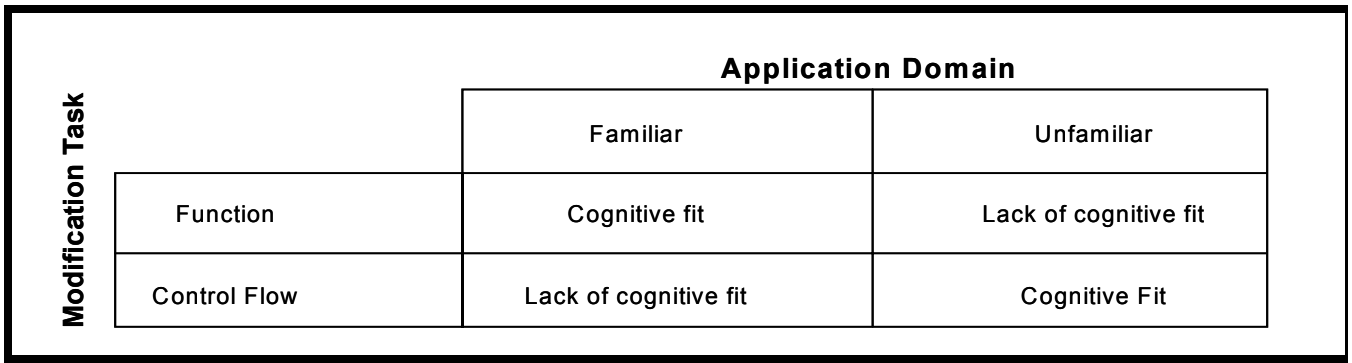

Figure 5. Research Design for Testing Cognitive Fit Between Familiarity with the Application Domain and Type of Modification

of the modification task are dissimilar, the two tasks are not mutually supportive; each task, therefore, demands specific attention, resulting in an inverse relationship between changes in software comprehension and performance on the modification task.

In summary, we operationalize cognitive fit by pairing familiarity with the application domain and the type of modification task. We test our hypotheses by measuring the changes in the maintainer's comprehension that occur during the conduct of the modification task. We hypothesize a three-way relationship between application domain familiarity, type of modification task, and changes in comprehension. Figure 5 presents the research design.

Research Method

To investigate the hypotheses, we conducted a study in which IT professionals studied and modified computer software. As specified above, combinations of application domain familiarity and modification task type create four conditions: two with cognitive fit, two without cognitive fit. A software maintainer conducted one of two types of modification tasks (function or control flow) in each application domain. Half of the participants completed a function modification in both application domains, while the others completed control flow modifications. Each participant worked in situations both with and without cognitive fit. To evaluate changes in comprehension, we created sets of comprehension questions that were administered before and after conducting each modification.

Task Setting

COBOL was the programming language for the experimental programs because of its predominance as a business programming language. An estimated 150 billion to 175 billion lines of COBOL code are in production worldwide with 83 percent of transactions processed by COBOL running on mainframes (Arranga 2000). COBOL applications, therefore, represent a significant corporate asset (Coyle 2000), making them particularly relevant to studies of software maintenance.

Accounting was selected as the familiar application domain because accounting systems are frequently implemented in COBOL, thereby increasing the likelihood of identifying potential subjects. Hydrology was selected as the unfamiliar application domain because it is unrelated to accounting and subjects familiar with accounting were unlikely to be familiar with hydrology.

Subjects

In total, 24 IT professionals employed in developing and maintaining COBOL accounting applications participated in the main study. The average participant was 36.5 years of age (range: 24 to 52), had 10.7 years professional IS experience (range: 3 to 20), knew 5 programming languages (range: 1 to 11), and was male in 4 out of 5 instances. The average self-reported score for COBOL knowledge was 4.7 on a five-point scale (range: 3 to 5). Job titles ranged from senior programmer to team leader. Accounting domain knowledge was assessed via the number of accounting credit hours (average = 10.2 credit hours) and the number of years of experience in accounting applications (average = 6.7 years). Participants' experience was examined to ensure that none had experience in hydrology or other scientific domains. Observations during data collection and comments from participants' supervisors suggest that participants were highly motivated to complete the task.

Experimental Materials

The experimental materials included computer programs, comprehension questions, and modification specifications. The computer programs served as stimulus materials, while the comprehension questions were used to assess a maintainer's comprehension of the

programs. The modification specifications presented requested changes to the software.

Computer Programs

Three programs were used in the experiment: one to familiarize participants with the experimental procedures, and one each from the accounting and hydrology domains. The programs accessed all files sequentially and contained internal sorts.

The practice program sorted an input inventory file and prepared a report that listed inventory levels for all items in a prescribed order. It consisted of 99 source lines of code (SLOC), with 33 SLOC in the PROCEDURE DIVISION.

The accounting program was a modified version of an operational payroll program that computed and produced paychecks, mailing labels, and a pay roster. The sort sequenced time cards by location, last name, and social security number. The pay roster was organized by location and listed the name, social security number, budget code, gross pay, and net pay of each individual who received a paycheck. Subtotals of net and gross pay for each location and overall totals were accumulated. We made modifications to the software to facilitate comprehension by using more meaningful paragraph and variable names, and reordering paragraphs. The program consisted of a total of 417 SLOC with 160 SLOC in the PROCEDURE DIVISION.

The hydrology program was a modified version of an operational water quality management program that computed averages and variances for seven parameters (coliform, nitrate, chloride, lead, fluoride, pH, and alkalinity) based on test results. A sort organized the test information by well identification number, year, and month. The report was organized by well identification number and contained the average, variance, and maximum value of test results for each month. We made modifications to improve comprehension similar to those made to the accounting program. We also revised the structure of the program, which had become difficult to understand due to changes made over time. The program consisted of a total of 416 SLOC with 162 SLOC in the PROCEDURE DIVISION.

For experimental purposes, it was important that the two programs be comparable. Assessing the comparability of software is complicated by the fact that hundreds, if not thousands, of software metrics exist (Zuse 1991). Further, no theory suggests which metrics are appropriate in specific instances. The confusion is complicated by the fact that a particular software metric typically focuses on only one of the many types of information embedded within a piece of software (usually control flow or data flow; see Zuse 1991).

Because there is no single accepted software metric, we employed three metrics to compare our experimental programs: SLOC, data density, and decision density. SLOC controls for size and is used to assess overall comparability. SLOC is highly correlated with most other metrics (Kemerer 1995). Further, it is as effective a measure of software comparability as other measures (Munson and Khoshgoftaar 1989). The major criticisms of SLOC as a metric are that it cannot be used to compare effort across programming languages and that different counting standards can invalidate within-language comparisons (Jones 1994). These criticisms are not relevant to this study because we use a single programming language and a consistent counting standard (see Jones 1986). Table 1 presents SLOC counts for the two experimental programs. Note that the programs are comparable with respect to the DATA and PROCEDURE DIVISIONS, as well as overall.

We also used data and decision density (see Table 1), two metrics used by Banker et al. (1998) to investigate maintenance. Data density is defined as Halstead's (1977) N2 software science measure divided by total lines of code. N2 is derived from a count of the number of operands (data elements) referenced in the software. Decision density is McCabe's (1976) cyclomatic complexity divided by total lines of code. Cyclomatic complexity is a measure of the number of decision paths in a piece of software. We used Banker et al.'s data as estimates for the standard deviations of data and decision density to conduct t-tests to determine if our experimental programs were significantly different.[3] The accounting and hydrology programs were not significantly different with regard to either data ($t = -.178$, $p = .86$) or decision density ($t = -.351$, $p = .73$).[4] Therefore, the two programs used in this study are comparable based on the measures of SLOC, data density, and decision density.

Comprehension Measure

To assess the relationship between comprehension and modification performance, we needed a measure of the software comprehension that occurs while maintainers are engaged in the modification task. We measured this difference by assessing comprehension prior to and following each modification task.

Apart from the need to measure the change in comprehension that occurs during modification, introducing an initial study period into the experiment served a number of other purposes. First, it strengthened our manipulation of software maintainers' reliance on a domain model when familiar with the application domain (Vans et al. 1999). An initial comprehension period did not influence the mental representations of those maintainers whose dominant *mental*

[3]Banker et al.'s study is based on the entire maintenance portfolio of a national mass-merchandising retailer. Because it is the only examination of an entire applications portfolio of which we are aware, it provides the best available estimates of standard deviation for data and decision density.

[4]T-tests were used because one can retain the null hypothesis of normality for data and decision densities ($p = .1914$ and .1612, respectively). We computed the t-statistics as follows: $t = (\text{metric}_{\text{domain1}} - \text{metric}_{\text{domain2}})/(\text{standard deviation} * \sqrt{2})$.

Table 1. Comparability Measures for the Original Programs

Program Characteristics	Familiar (Accounting)	Unfamiliar (Hydrology)	p-value
SLOC			
Identification Division	3	3	
Environment Division	20	18	
Data Division	234	233	
Procedure Division	150	162	
Total	*417*	*416*	
Data Density Total Operands (N2/Total SLOC)	.6787	.7308	.86
Decision Density Cyclomatic Complexity (VG1/Total SLOC)	.0312	.0432	.73

representation was the program model because some prior knowledge of the software should not alter the likelihood that maintainers who are unfamiliar with the application domain will rely upon their program model (Vans et al. 1999). Second, an initial comprehension period followed by a distinct modification period is consistent with previous comprehension research (Corritore and Wiedenbeck 1999; Pennington 1987b) and allows us to examine alternative explanations for our findings. Third, it is similar to practical situations where a software maintainer has some experience with a piece of software prior to conducting a modification. Finally, this approach creates a measure of comprehension relative to each maintainer's comprehension skills; that is, it controls for individual differences in the effectiveness of comprehension.

Two approaches have been used predominantly to assess programmers' comprehension of computer programs: free recall (Sheppard et al. 1979; Sime et al. 1973) and responses to comprehension questions (Corritore and Wiedenbeck 1999; Curtis et al. 1989, Green 1977, Pennington 1987a, 1987b). Free recall requires a programmer to recall a program after a period of study. It requires memorization, which may or may not involve understanding (Boehm-Davis 1988; Curtis et al. 1989), and can be used for only quite small programs. Because we wanted to assess comprehension rather than memorization, we used comprehension questions. Further, comprehension questions frequently have been employed to assess a maintainer's *mental representation of software* (see Corritore and Wiedenbeck 1999; Pennington 1987b).

Our assessment of software comprehension required the preparation of two sets of comprehension questions for each application domain, one set each to be administered prior to and following the modification task. Presentation of the question sets was counterbalanced so that half of the participants responded to one set following the study period, and the other set following the modification period.

To be consistent with previous assessments of comprehension (Corritore and Wiedenbeck 1999; Pennington 1987a, 1987b), we developed questions based on each of the different types of information: function, data flow, control flow, and state (Pennington 1987a, 1987b) in the following series of steps. First, we represented each program using four documentation formats, each of which corresponded to a type of information: functional decomposition to represent function information, data flow diagrams to represent data flow information, flowcharts to represent control flow information, and decision tables to represent state information. Second, based on example questions in Pennington (1987a), we used the documentation formats to develop questions relating to each type of information. Third, we then randomly assigned a question to one of the two question sets, with five questions relating to each type of information, and with 10 "yes" and 10 "no" responses, yielding 20 questions in each set. The sequence of questions with respect to the type of information was randomly generated and the same for each set.

We then evaluated our comprehension instruments in a further series of steps. Two knowledgeable colleagues who were not involved in the research reviewed the questions. The questions were then tested in a pilot study whose participants included professional software maintainers, resulting in minor wording changes. We assessed the reliability of the question sets using the Kuder-Richardson statistic (the special form of Cronbach's Alpha for dichotomous variables) using the data from the main experiment. The average alpha was .72 (accounting domain: .75 and .60; hydrology domain: .83 and .71), which meets conventional levels of acceptability and compares favorably with the values reported in the only other comprehension study of which we are aware that reported reliabilities (Curtis et al. 1989). Further, the accuracy of programmers' responses did not differ statistically for the two question sets in each domain ($p > .05$). Hence, the two sets of questions in each domain were essentially similar and we could use them to measure comprehension before and after programmers conducted modifications in those domains.

Examples of questions representing each knowledge category are presented below.[5] The first two questions are from the familiar (accounting) domain, and the final two from the unfamiliar (hydrology) domain. The first and third questions are correctly answered by "yes," the second and fourth by "no." Comprehension questions typically require the software maintainer to translate their knowledge into application terms (see the first question, which is a function type item) or to integrate their understanding over different elements of the program (the remaining questions). Hence they require understanding beyond rote memorization.

- Are mailing labels created for each employee?
- Does the value of PARAM-TEST affect the value of TIME-MAX?
- Is OUTPUT-VALUES performed before OUT-MONTH-HEADERS?
- Does LOC-DL equal NEW KENSINGTON when ST-LOCATION equals UP?

Initial and final comprehension scores were created by converting the number of correctly answered questions into a percentage, consistent with Pennington (1987b) and Corritore and Wiedenbeck (1999). We then computed a score based on the percent change in comprehension. Note that simple difference scores (posttest–pretest) place those who achieve a high initial score at a disadvantage: due to higher levels of initial comprehension, they cannot demonstrate as much improvement between the pre- and post- tests. Therefore, our percent change in comprehension score calculates the percent change as the percent of possible improvement.[6] If a maintainer demonstrated higher levels of comprehension after conducting the modification task, the comprehension score indicates the percent improvement of the possible improvement as follows:

(final comprehension score – initial comprehension score)/(100 – initial comprehension score)

We followed similar logic to develop scores for those who demonstrated a decrease in comprehension after conducting the modification task. The formula computes the percent decrease in comprehension of the possible decrease:

(final comprehension score – initial comprehension score)/initial comprehension score

Note that the range of possible values is consistent for both formulas: 0 to 100 percent and positive or negative depending upon whether the software maintainer demonstrated an increase or a decrease in comprehension. Responses to the comprehension questions were scored by a secretarial assistant and were not available to the researchers during scoring of the modification tasks.

Modification Tasks

Tasks that emphasized control flow and function information were required for both application domains. The control flow tasks required maintainers to insert a new level of control break into an existing report. As noted in our specifications (Appendix A), maintainers were provided with copies of the existing and required (modified) outputs so that they could easily determine that the modification was a change to an existing report. Control flow tasks require maintainers to engage mentally with the details of the current program's execution sequence and make enhancements to support the additional level of control break. Such a task emphasizes the program's execution sequence, which is consistent with the dominant *mental representation of the software* (program model) in the unfamiliar domain. Control flow modification tasks, therefore, create conditions of fit in the unfamiliar (hydrology) application domain and no fit in the familiar (accounting) application domain.

The function tasks required software maintainers to alter the existing programs by incorporating information from a new input file to create and output new information. Function tasks require maintainers to understand the higher level goals of a program. Such tasks emphasize information such as program goals or functional units of the existing program rather than implementation details. Function tasks, when paired with application domain familiarity, create opposite fit conditions from those for control flow tasks; that is, function tasks create conditions of fit in the familiar (accounting) domain and no fit in the unfamiliar (hydrology) domain.

In addition to operationalizing control flow and function type tasks, two criteria were used to develop the modifications. First, the modification should represent a realistic task that could be approached within our time constraints. Second, the resulting programs should be comparable. We again used three metrics to assess software comparability. It was important that the modifications be of consistent size because the number of lines of code added during a maintenance task is highly predictive of the effort to conduct the modification (Sheppard et al. 1979); otherwise, differences in performance could be attributable to inconsistent levels of effort required to conduct a modification. Our primary size criterion was overall size, followed by size within type of modification task (i.e., control flow or function) because a maintainer conducted the same type of modification task in both application domains to create both cognitive fit and no-fit conditions. Table 2 presents the counts of SLOC required to implement the modifications. Because the necessary changes to the input and output record formats were provided to the participants, these changes are not reflected in the counts (see the "Pilot Study" section below for further information). The data and decision densities for both types of modifications also indicate that they are comparable (Table 3 compares the modifications by application domain and by task type).

[5]The complete sets of comprehension questions are available from the first author upon request.

[6]Note, however, that the simple difference score (posttest–pretest) yields essentially similar results for hypothesis testing and analyses of alternative models.

To assess participants' perceptions of the difficulty of a modification, after completing each modification task participants were asked: "How difficult was it to develop the modification?" They responded on a Likert-style scale with 1 = very easy and 7 = very difficult as the anchors. There were no statistically significant differences between perceptions of the two types of modification tasks (function = 3.13 versus control flow = 3.54, $F_{1,22} = .68$, $p = .42$). Hence, based on both software metrics and participants' perceptions, the modification tasks appear to be of similar difficulty.

Experimental Design

A mixed design with application domain familiarity as the within-subjects factor and type of modification task as the between-subjects factor was used to investigate the hypotheses. By varying application domain familiarity and type of modification task in this manner, each software maintainer worked in both fit and no-fit conditions. As such, each subject served as their own control for level of programming expertise. Further, by fully crossing two types of modification tasks with two levels of application domain familiarity, we controlled for the possibility that our findings are idiosyncratic to a particular modification task.

We controlled for application domain familiarity by prescreening participants. A further check on application domain familiarity was made by asking participants to rate their familiarity with the application domain after the initial comprehension period. A paired t-test of differences in responses indicated that participants found the accounting domain more familiar than the hydrology domain ($t_{23} = 5.84$; $p < .0001$). The order of presentation of the two application domains was counterbalanced so that half of the participants worked in the familiar application domain first, then the unfamiliar domain, and vice versa. Participants were randomly assigned to one of the two types of modification tasks.

Pilot Study

We conducted a pilot study to test the experimental materials and procedures, and to determine the length of the initial comprehension period. Students who had completed a COBOL course served as pilot subjects initially. Based on our observations, it was clear that our manipulation of application domain familiarity was ineffective as students had insufficient background to be sufficiently familiar with either application domain for our purposes. Therefore, for the later stages of pilot testing and the main experiment, professional software maintainers experienced with accounting applications served as participants.

Early in our study, we decided to restrict the length of the entire experiment to approximately 4 hours because we regarded 4 hours as the upper limit for the participation of professionals whose time was donated by their employers. The pilot study enabled us to determine the length of time that needed to be devoted to various phases of our experiment. As a result, software maintainers in the main study were permitted 15 minutes for the initial comprehension period and 35 minutes for the modification period in each application domain. Note that allowing unlimited time to conduct the modification task could have created an unnatural setting (Walz et al. 1993). Further, allowing unlimited time would have created a confound with our measure of comprehension because different maintainers would have worked with the software for unequal time periods.

The initial comprehension period avoided "ceiling effects" (Sheil 1981), while allowing participants sufficient time to study the source code. Most participants had ample time to review the source code and were studying it for a second time when requested to stop. The length of the modification period was sufficient to allow most participants to design the modification and begin making changes to the code. In general, subjects were able to complete all elements of the experiment within four hours. A few maintainers took slightly longer due to the time they took completing the questionnaires or questions they had during the practice session.

Our pilot study also revealed that making the changes to the DATA DIVISION proved to be extremely time-consuming. This process is somewhat mechanical and did not provide insight into maintainers' ability to conduct the meaningful elements of the modification task. Therefore, the necessary changes to the input and output record formats were made to the program listings in the DATA DIVISION, but not in the WORKING-STORAGE section. These versions were compiled to provide cross-reference listings and the changes made to the DATA DIVISION were highlighted to aid the participants. The original and modified listings were provided to maintainers at the beginning of the modification phase. These changes were made early in the pilot study, tested on later pilot subjects, and judged effective. Some participants commented that the procedure was similar to having an analyst work out formatting on a printer spacing chart prior to giving a task to a programmer. This adjustment markedly increased the progress made on the modification task in the allotted time.

Experimental Procedure

The first author ran the subjects individually through the study. The experiment took place in three segments: a practice session (to familiarize participants with the experimental procedures), followed by two experimental sessions (one in each application domain). The process was the same for all sessions, although the time permitted differed (see below). Participants were allowed a short break following each session.

Each segment consisted of an initial comprehension period and a modification phase. At the beginning of a task, the participant was given a copy of the source code and asked to study the software to gain as great an understanding of the software as possible in the

Table 2. Comparability Measures for Modified Programs

Program Characteristics	Function Modification		Control Flow Modification	
	Familiar (Accounting)	Unfamiliar (Hydrology)	Familiar (Accounting)	Unfamiliar (Hydrology)
SLOC—PROCEDURE DIVISION				
Modified Program	185	185	192	196
Original Program	(160)	(162)	(1060)	(162)
SLOC added to PROCEDURE DIVISION	25	23	32	34
Additions to WORKING STORAGE SECTION	2	1	7	5
SLOC changed in PROCEDURE DIVISION	1	–	–	–
Total SLOC added or changed	28	24	39	39
SLOC Additions to Record/File Formats (Provided to Participants)	22	9	19	4
Data Density	0.7326	0.8062	0.7478	0.8505
Decision Density	0.0404	0.0432	0.0329	0.0436

Table 3. Comparison of Data Density and Decision Density for the Modified Programs

	Familiar (Accounting)				Unfamiliar (Hydrology)			
	Function	Control Flow	t value	p value (2 tailed)	Function	Control Flow	t value	p value (2 tailed)
Data Density	0.7326	0.7478	–0.0521	0.9588	0.8068	0.8505	–0.1497	0.8821
Decision Density	0.0404	0.0329	0.2191	0.8281	0.0432	0.0436	–0.0117	0.9908
	Function				**Control Flow**			
	Familiiar (Accounting)	Unfamiliar (Hydrology)	t value	p value (2 tailed)	Familiar (Accounting)	Unfamiliar (Hydrology)	t value	p value (2 tailed)
Data Density	0.7326	0.8068	–0.2542	0.8012	0.7478	0.8505	–0.3518	0.7276
Decision Density	0.0404	0.0432	–0.0818	0.9354	0.0329	0.0436	–0.3126	0.7569

amount of time allocated (5 minutes for the practice program, 15 minutes for each program in the study proper). After the initial comprehension period, the participant responded to one set of comprehension questions with no explicit time limit and without access to the source code. They did not have access to the source code because we wanted to assess their knowledge as represented in their *mental representation of the software*.

In the modification phase, the participant was given the source code they had been studying, the one-page modification specification, the modified hard-copy listing that contained the changes to input or output record formats, and any relevant input or output files. The participant was asked to work on the modification task for the given amount of time (10 minutes for the practice program, and 35 minutes for the study proper). During the practice session, the researcher showed the participant how to compile, link, and run the modified practice program, as well as how to print and display the output generated by the program. During the study proper, the experimenter answered only technical questions that were unrelated to the conduct of the experimental tasks. At the end of the modification phase, the source code and other materials relating to the experimental task were again removed and the participant responded to the second set of comprehension questions.

Assessment of Performance on the Modification Task

To assess performance on the modification tasks, objective scoring criteria were developed for each modification by identifying several subtasks, some of which were further subdivided.

Each subtask was assigned a specific point value based on its importance in accomplishing the modification, for a total of 100 points. When a participant did not make the changes related to a subtask in the online version of their program, the researcher examined the hard-copy program listing the participant used during the experiment for handwritten changes. Up to 75 percent of the points for a particular subtask were awarded, depending on the completeness of such changes. The total score for each modification task was the sum of the points earned from the changes made to the online version and the hand-written changes to the hard-copy listing. Points were assigned to give maintainers credit for all the relevant work that they had accomplished; hence, they could earn partial credit for any identified subtask. The first author used this scoring scheme to assign points to the modifications conducted by the participants (see Appendix B).

The scoring criteria, therefore, emphasized the amount of progress made toward completing the task. The modifications required fairly standard programming approaches and there were no substantive differences in the quality of the modifications; that is, all programmers displayed a professional approach to preparing their solutions and we were unable to detect much variance beyond the amount of work accomplished. We did not explicitly deduct for errors (of which there were few) because any effort expended on an incorrect approach would have been at the expense of making the requested changes and would have resulted in double penalization.

To assess the validity of the scoring criteria, the scores were compared with two sets of rankings. The first author and a research assistant, who had two years' professional experience in COBOL development and maintenance and no knowledge of the research hypotheses, ranked a subset of five of the modified programs for each task type in each domain. The rankings were based on each rater's subjective assessment of how well each maintainer had progressed on the modification. The two sets of rankings were then correlated with the ranks of the scores from the objective assessments (within each domain by task combination). Correlations ranged from .82 to 1.00 and all were significant at $p < .05$. Satisfied by the high degree of agreement between the objective scoring and subjective rankings, we used the objective scores as the modification performance measure in this study.

Analysis and Results

We first present our analyses and findings with respect to the theoretical development proposed in this paper. We then strengthen the support for our findings by presenting the results of analyses designed to consider alternative explanations of the results.

Analyses Based on Theoretical Model

The data were analyzed via SAS PROC MIXED, which allowed us to specify both within- and between- subjects factors and continuous variables in the within-subjects effect (Wolfinger and Chang 1995). PROC MIXED was required because percent change in comprehension was measured twice for each participant (once in each application domain). Hence, we required a technique that allowed us to specify percent change in comprehension for both observations of each participant. The model specifies performance on the modification task as the dependent variable and three independent variables (application domain familiarity, type of modification task, and percent change in comprehension). Two of the independent variables were manipulated in the experiment to form the cognitive fit conditions: application domain (familiar and unfamiliar), a within-subjects factor, and type of modification task (function or control flow), a between-subjects factor. The final independent variable, percent change in comprehension, was specified as a continuous variable specific to each participant in both application domains.

Table 4 presents the means and standard deviations of the percent change in comprehension and modification scores in each of the four experimental conditions. From an examination of the means for percent change in comprehension across the two modification task types, one might be concerned that there could be a confound due to a relationship between these two independent variables. Further analysis revealed, however, that there is no statistically significant difference in percent change in comprehension based on modification task type. Further, a model that included percentage change in comprehension as a dependent variable with the experimental factors (application domain familiarity and type of modification task) did not yield a statistically significant fit. Hence, these variables do not create a confound in our subsequent analysis.

We analyzed the data in three stages: (1) the main effects exclusively (application domain familiarity, type of modification task, and percent change in comprehension); (2) main effects and two-way interactions; and (3) the full model (main effects and the two-way and three-way interactions). This approach allowed us to test if the addition of the two-way and then the three-way interactions significantly improved model fit. If this is not the case, then the simpler model is preferable (Tabachnick and Fiddell 2000).

Table 5 presents the results of our analyses. The model with three main effects was significant overall (i.e., the null model likelihood test) and the only significant main effect was for application domain familiarity. The addition of the two-way interactions resulted in a model that was significant overall; however, no factors were significant. Nonetheless, the model with the two-way interactions demonstrates a significantly better fit to the data than the main-effects only model. The full model (main effects and the two-way and three-way interactions) was significant overall and the hypothesized three-way interaction (application domain familiarity × type of modification task × percent change in comprehension) was significant ($F_{1,22}=8.77$, p-value = .007). Further, the full model results in a significantly better fit to the data than the model with the main effects and two-way interactions.

Table 4. Means and (Standard Deviations) for Percent Change in Comprehension and Modification

Type of Modification	Familiar (Accounting) Domain		Unfamiliary (Hydrology) Domain		Means	
Percent Change in Comprehension						
Function	.16	(.27)	.06	(.23)	.11	(.25)
Control Flow	.24	(.30)	.28	(.28)	.26	(.28)
Means	.20	(.28)	.17	(.28)	.19	(.26)
Performance on Modification Task						
Function	50.75	(23.25)	39.92	(25.84)	45.33	(17.61)
Control Flow	54.08	(18.66)	38.00	(31.67)	46.04	(24.24)
Means	52.42	(20.69)	38.96	(28.28)	45.69	(20.73)

Table 5. Results of Analysis[a]

Effects	Main Effects		Main Effects and Two-Way Interactions		Main Effects and Two-Way and Three-Way Interactions	
	f-value	p-value	f-value	p-value	f-value	p-value
Percent Change in Comprehension	1.93	.179	.85	.366	.69	.414
Type of Modification Task	.00	.973	.28	.599	.03	.864
Application Domain Familiarity	5.81	.025	.25	.620	2.13	.159
Percent Change in Comprehension × Type of Modification Task			1.77	.197	.02	.881
Percent Change in Comprehension × Application Domain Familiarity			3.02	.096	4.01	.058
Type of Modification Task × Application Domain Familiarity			.00	.957	3.94	.060
Percent Change in Comprehension × Type of Modification Task × Application Domain Familiarity					8.77	.007
Null Model Likelihood Test	$\chi^2 = 7.43$	p = .024	$\chi^2 = 9.45$	p = .009	$\chi^2 = 9.63$	p = .008
–2 Residual Log Likelihood	411.1		384.3		366.8	
Δ(–2 Residual Log Likelihood)			$\chi^2 = 26.8$	p < .001	$\chi^2 = 17.5$	p < .01

[a]Tests to determine if the inclusion of the interactions terms yields a model with an improved fit are conducted by computing the difference of the –2 residual log likelihood values for the two models, which is distributed as χ^2 with df equal to the difference in the number of parameters for the two models.

The percentage change in comprehension variable is distributed in a non-normal fashion. We therefore replicated the analysis using rank transformed values for the percentage change in comprehension and modification scores. Conover (1999) recommends this procedure to address concerns about a data set meeting the assumptions of a parametric test, including the possible influence of outliers. The analysis of the ranked data confirmed the results obtained from the parametric analysis—specifically, the three-way interaction is significant ($F_{1,22} = 8.62$, $p = .008$). In these circumstances, one can conclude that the departure from normality is not sufficient to violate the assumptions of the model and that the results of the parametric analysis are valid (Conover 1999).

The detection of the three-way interaction is essential to the process of assessing support for our hypotheses. Specifically, because of the complete crossing of the experimental factors, the significant three-way interaction indicates that the slopes of the lines in the cognitive fit conditions are significantly different from the slopes in the no-fit conditions; that is, the relationship between percent change in comprehension and performance on the modification task differs from the fit conditions to the no-fit conditions.

To further determine if the significant interaction is consistent with our hypotheses, we examined the correlations between percent change in comprehension and modification performance for each experimental condition. As we indicate in Figure 6, and consistent with our hypotheses, in the cognitive fit conditions (familiar application domain/function modification and unfamiliar application domain/control flow modification), the relationships between percent change in comprehension and modification performance are positive. When cognitive fit does not exist (familiar application domain/control flow modification and unfamiliar application domain/function modification) the relationships are negative. In conditions of cognitive fit, therefore, increases in comprehension of the software are associated with higher levels of performance on the modification tasks. Conversely, when cognitive fit does not exist, increases in comprehension are associated with lower levels of modification performance.

To depict visually the nature of the relationship between percent change in comprehension and performance on the modification task, we fitted lines to the observations in each experimental condition. Figure 7 displays the relationship in the two cognitive fit conditions (familiar application domain/ function modification and unfamiliar application domain/ control flow modification). Both lines display an upward trend, indicating a direct relationship between percent change in comprehension and performance on the modification task, consistent with the correlations reported in Figure 6. Figure 8 displays the relationship in the two conditions in which cognitive fit does not exist (familiar application domain/control flow modification and unfamiliar application domain/function modification). Both lines display a downward trend, indicating an inverse relationship between percent change in comprehension and performance on the modification task, consistent with the correlations reported for those conditions in Figure 6.

Hence all of our hypotheses are supported, H1a and H1b for a positive relationship between increases in comprehension during modification and performance on the modification task when cognitive fit exists and H2a and H2b for an inverse relationship between increases in comprehension during modification and performance on the modification task when cognitive fit does not exist.

Examination of Figures 7 and 8 also reveals that some maintainers experienced decreases in comprehension during the modification process. Further investigation shows that a total of 10 observations (distributed over 9 maintainers) demonstrated negative percent changes in comprehension. These changes occurred in all experimental conditions with average decreases ranging from 14 to 19 percent. We believe that the maintainers may have focused their attention on what they perceived they needed to comprehend in order to complete the modification task and, as a result, were not then able to recall some of the information that they did not revisit during their second exposure to the software. Hence, the fact that this second exposure resulted in lower comprehension than the first reflects the more focused nature of their interaction with the software.

Analyses Based on Possible Alternative Explanations

The prior analysis is based on a measure of the change in comprehension that occurs while a software maintainer is modifying the software. To lend strength to our theoretical formulation, we consider two alternative explanations for our findings. First, a software maintainer who develops a good initial understanding of the software may be more effective at conducting the modification. Second, performance on the modification task may reflect the maintainer's understanding of the software after conducting the modification task.

We examined the first alternative by conducting the same analysis as presented above, with the maintainer's score on the initial administration of the comprehension questions replacing the percent change in comprehension score. If higher initial comprehension levels were associated with higher levels of performance on the modification task, the score for initial comprehension should be significant in the analysis. When we reanalyzed the data, each model was significant ($p < .05$), but none of the factors, including that for initial comprehension, was significant in any of the analyses. For all models, the analyses based on percent change in comprehension (the initial analysis) demonstrate a better fit to the data than the corresponding model using the initial comprehension scores.[7]

[7]The test used to determine if the addition of interaction terms creates a statistically significantly better fit can be used to compare models that are subsets of one another, but cannot be generalized to compare competing models with the same number of parameters. Hence, although the fit indexes are considerably lower (better) for the analysis based on the percent changes in comprehension, we were unable to test for statistically significant differences.

Modification Task	Application Domain: Familiar	Application Domain: Unfamiliar
Function	**Cognitive fit:** positive relationship between software comprehension and performance on modification task **(r = .52)**	**Lack of cognitive fit:** negative relationship between software comprehension and performance on modification task **(r = -.30)**
Control Flow	**Lack of cognitive fit:** negative relationship between software comprehension and performance on modification task **(r = -.35)**	**Cognitive fit:** positive relationship between software comprehension and performance on modification task **(r = .49)**

Figure 6. Observed Relationships Among Cognitive Fit, Comprehension, and Modification Performance

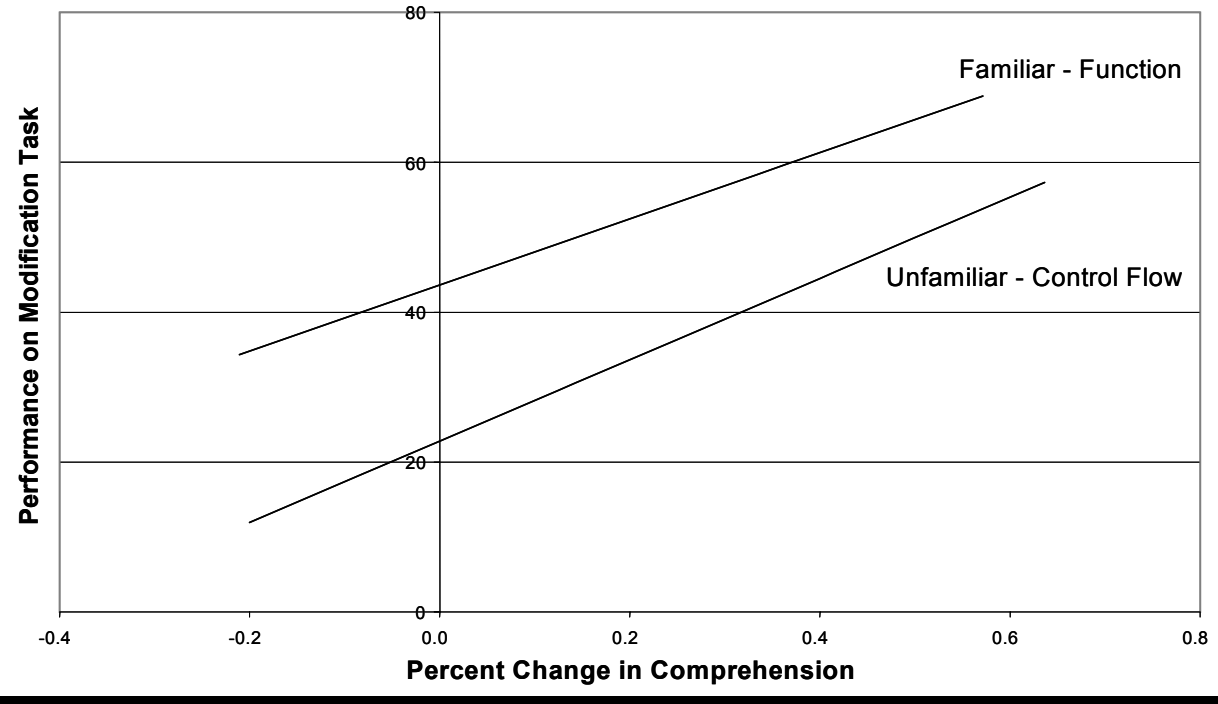

Figure 7. Relationship Between Percent Change in Comprehension and Performance on Modification Task in Conditions of Cognitive Fit

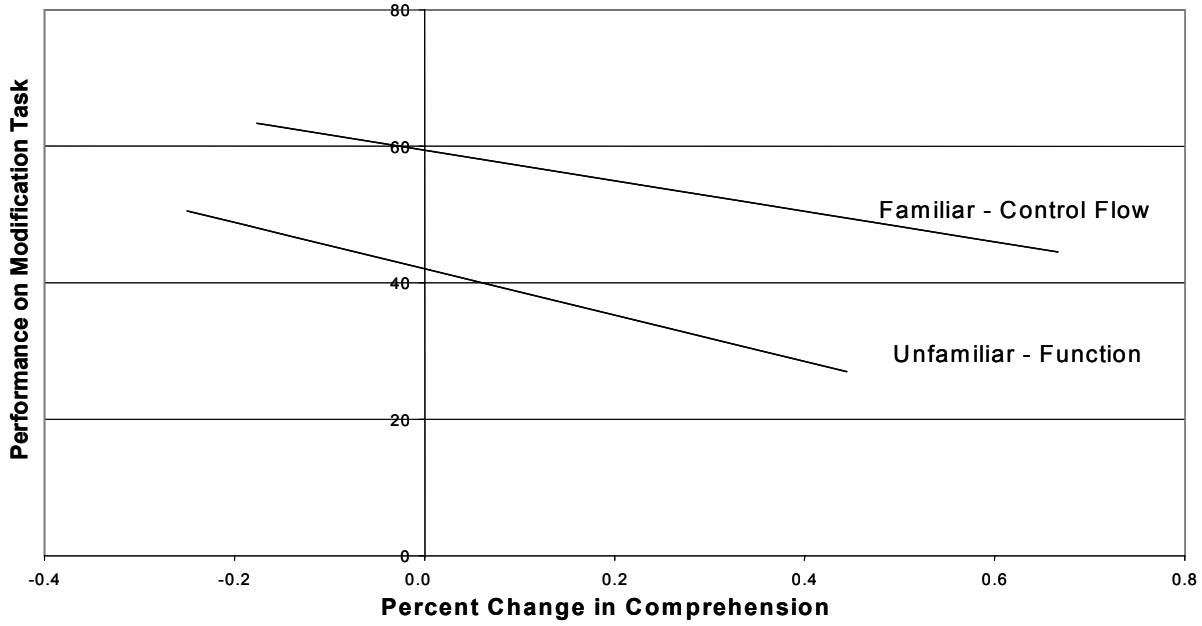

Figure 8. Relationship Between Percent Change in Comprehension and Performance on Modification Task in Conditions Where Cognitive Fit Does Not Exist

Hence, a maintainer's initial comprehension does not appear to explain *problem-solving performance* on the modification tasks.

To consider the second alternative, we replaced our percent change in comprehension score with the maintainer's score on the final administration of the questions. If higher final comprehension levels were associated with higher levels of performance on the modification task, the score for the final administration of the comprehension questions should be a significant factor in the analysis. The analysis that included the main effects only was significant overall, but none of the individual factors was significant. For the two analyses that include interaction terms, the tests for the overall fit of the model were not significant, nor were any of the factors. For each model, the analyses based on percent change in comprehension demonstrate a better fit to the data than the corresponding model using the final comprehension score. Hence, a maintainer's final comprehension does not appear to explain *problem-solving performance* on the modification tasks.

In summary, the results of the additional analyses indicate that the most insightful understanding of a software maintainer's performance on software modification tasks is based on the changes in comprehension that occur while conducting a modification task rather than initial or final levels of comprehension.

Discussion and Implications

This section discusses the research findings and implications for future research and practice.

Discussion of the Findings

Our overarching contribution in this study is to demonstrate the interrelationships between software comprehension and modification tasks during software modification and, as a consequence, to motivate the need to study software tasks other than comprehension alone.

The results of our analyses support our hypotheses that improved software comprehension is associated with better performance on the modification task only when cognitive fit exists between the maintainer's dominant *mental representation of the software* and the type of modification task conducted (*software modification task*). When fit does not exist, improved software comprehension is associated with lower *problem-solving performance* on the modification task. Our experimental manipulation of fit and no-fit conditions creates a fully crossed design, which provides a strong test of the model because it rules out the possibility that a single experimental condition (i.e., familiarity with the application domain

or type of modification task) explains our results. Furthermore, because each participant worked in both fit and no-fit conditions and we detected an inverse relationship between changes in comprehension and performance on the modification task only in the two no-fit conditions, we can attribute this finding to the lack of cognitive fit between the maintainer's dominant *mental representation of the software* and the type of *software modification task*. In addition, we also considered, and ruled out, two alternative explanations for performance on the modification task, providing greater confidence in the robustness of our results and underlying theory (Stinchcombe 1968).

While our sample size of 24 software professionals may appear to be a limitation, our experiment uses a mixed, within-between subjects design that yields 48 observations. As a further check on the adequacy of our sample size, we repeated our analysis using the Hotelling-Lawley-McKeon and Hotelling-Lawley-Pillai-Samson *F* approximations, both of which are recommended over the default *F* statistic for small samples. In both cases, the analysis results in *F*- and *p*-values identical with those reported in our primary analysis, suggesting that our sample is not small in the statistical sense. Furthermore, only professional software maintainers with a minimum of 2 years of experience served as participants. We examined participants' backgrounds regarding the types of applications with which they had experience. Other than familiarity with accounting applications and no experience with scientific applications, as necessary for our experimental manipulations, we could find no systematic differences or commonalities that would lead us to question the representativeness of our sample.

A second potential limitation of our study is that we tested our theory via the emergent effects of our theoretical model; that is, rather than examining *mental representations for task solution* directly, we assessed *problem-solving performance*. In other words, we used the approach taken by all studies that use the theory of cognitive fit as their theoretical foundation.

Our study also makes a number of specific contributions to the literature. First, our findings are particularly important because they run counter to the prevailing wisdom that high levels of comprehension necessarily lead to improved performance on related software tasks.

Second, our findings also suggest that software comprehension studies that use tasks such as maintenance and debugging simply as a way of allowing developers to interact with the software in order to assess the effects on software comprehension would be significantly richer if they examined performance on the *secondary* task as well as on the comprehension task. For instance, in addition to addressing how different factors influence comprehension, studies could investigate how such factors influence the performance on the related task as well as the relationship between comprehension and task performance.

Third, the present study extends the research on cognitive fit in two ways. First, the original conceptualization of cognitive fit is extended to consider its role in moderating the relationship between the performance on two interrelated tasks, the demands for which may potentially conflict. This conceptualization is based on the notion of dual-task interference. Second, based on the theory of distributed cognition (Zhang and Norman 1994), we incorporate the maintainer's knowledge of software and software development into the model of cognitive fit as factors that contribute to the software maintainer's dominant *mental representation of the software*. Such knowledge is developed over time as a result of prior experiences with, and therefore knowledge of, software.

Implications for Research and Practice

Studies of software maintenance are important to the information systems field because of the enormous resources companies devote to maintenance. Prior research on software maintenance has focused largely on software comprehension, perhaps because comprehension is generally regarded as underlying the majority of maintenance tasks. In this research, we take the first steps toward initiating a new view of the role of software comprehension in maintenance by focusing on the interaction between comprehension and modification tasks.

From the viewpoint of future research, our work suggests a number of further studies. In assessing *problem-solving performance*, we examined the outcome of the problem-solving process rather than examining the process directly. Future research could examine the process more directly by using, for example, protocol analysis.

Studies that examine the relationship between comprehension and modification more directly are needed. Protocol analysis could also provide insight into situations of both fit and non-fit. Such studies could investigate more directly the mechanisms of dual-task interference and help to answer questions such as: In conditions in which fit exists, are software maintainers able to gain just the knowledge needed for the modification, thus allowing them to better allocate their attention between comprehension and modification? Or, does the consistency between their dominant *mental representation of the software* and the modification task (*software modification task*) allow them to switch between the subtasks of comprehension and modification more efficiently? In conditions in which fit does not exist, it appears that some maintainers may overemphasize comprehension at the expense of modification. Are they unable to allocate their attention between comprehension and modification; that is, is the additional knowledge gained during comprehension simply irrelevant to the modification? When cognitive fit did not exist, some maintainers were relatively more successful at conducting the modification, What cognitive processes allowed them to overcome the difficulties of the mismatch?

Another important area for future research is to examine performance on software tasks other than maintenance tasks, and modification tasks other than enhancements. Many software tasks involve multiple subtasks that might also conflict with each other.

For example, software design requires a software developer to both understand the design requirements and translate those requirements into a solution plan. The representation used to document the requirements may be more or less consistent with an appropriate solution plan, potentially creating dual-task interference.

From the viewpoint of practice, a detailed understanding of the influence of cognitive fit on the interplay of comprehension and modification processes would allow researchers to provide better advice to software maintainers regarding approaches to software maintenance. For example, it appears advantageous for the maintainer to allow the requirements of a modification task to drive comprehension activities rather than attempting to fully understand software before initiating a modification. Such information should also be introduced into software development and software engineering education and training.

Given that it would be difficult to control for software maintainers' existing or dominant *mental representations of the software* and the types of *software modification task* to which they must respond, practice could focus on creating tools that help maintainers identify the knowledge necessary to complete the modification (in our study, control flow or function). Note that maintainers may need to be guided in their choice of appropriate tools because they might otherwise choose tools that highlight information consistent with their dominant *mental representation of the software* rather than the information needed to conduct the modification. For example, it might be possible to use a reverse engineering tool to represent a computer program at a higher level (arguably closer to the application domain) to help maintainers adopt a domain model as their dominant *mental representation* rather than a program model.

Conclusion

Prior research investigating software comprehension and modification views them as distinct tasks. Our findings indicate that they should be viewed as interrelated tasks because of the complex interrelationship between them. Seeking high levels of comprehension as a way of improving the ability to conduct other software-related tasks is beneficial when the software maintainer's *mental representation of the software* and their *mental representation of the modification task* emphasize the same type of knowledge. However, when there is a mismatch between the software maintainer's *mental representation of the software* and the *mental representation of the modification task,* improvement in comprehension impedes performance on the modification task.

Acknowledgments

The authors wish to thank Bob Zmud, Al Schwarzkopf, Shaila Miranda, Robert L. Glass, and Mark Sharfman for their insightful comments on earlier drafts of this manuscript. The authors also express their gratitude to Sandra Slaughter for providing access to the data used to estimate standard deviations for data and decision density.

References

Arranga, E. C. "In Cobol's Defense," *IEEE Software* (17:2), March/April 2000, pp. 70-72.

Banker, R. D., Davis, G. B., and Slaughter, S. A. "Software Development Practices, Software Complexity, and Software Maintenance Performance: A Field Study," *Management Science* (44:4), April 1998, pp. 433-450.

Barry, E., Kemerer, C., and Slaughter, S. A. "Toward a Detailed Classification Scheme for Software Maintenance Activities" in *Proceedings of the 1999 Americas Conference,* W. D. Haseman and D. L. Nazareth (eds.), Milwaukee, WI, August 1999, pp. 726-728.

Boehm-Davis, D. A. "Software Comprehension" in *Handbook of Human-Computer Interaction,* M. Helander (ed.), Elsevier Science Publishers, Amsterdam, 1988, pp. 107-121.

Boehm-Davis, D. A., Holt, R. W., and Schultz, A. C. "The Role of Program Structure in Software Maintenance," *International Journal of Man-Machine Studies* (36), 1992, pp. 21-63.

Broadbent, D. E. *Decision and Stress*, Academic Press, London, 1971.

Brooks, F. "No Silver Bullet," *IEEE Computer* (4:4), April 1987, pp. 10-19.

Burkhardt, J., Détienne, F., and Wiedenbeck, S. "Object-Oriented Program Comprehension: Effect of Expertise, Task and Phase," *Empirical Software Engineering* (7), 2002, pp. 115-156.

Conover, W. J. *Practical Nonparametric Statistics*, John Wiley and Sons, Inc., New York, 1999.

Corritore, C. L., and Wiedenbeck, S. "Mental Representations of Expert Procedural and Object-Oriented Programmers in a Software Maintenance Task," *International Journal of Human-Computer Studies* (50), 1999, pp. 61-83.

Coyle, F. P. "Legacy Integration—Changing Perspective," *IEEE Software* (17:2), March/April 2000, pp. 37-41.

Curtis, B., Sheppard, S., Kruesi-Bailey, E., Bailey, J., and Boehm Davis, D. "Experimental Evaluation of Software Documentation Formats," *The Journal of Systems and Software* (8), 1989, pp. 167-207.

Daly, J., Brooks, A., Miller, J., Roper, M., and Wood, M. "Evaluating the Effect of Inheritance on the Maintainability of OO Software," in *Empirical Studies of Programmers: Workshop 6*, W. D. Gray and D. A. Boehm-Davis (eds.), Ablex Publishing, Norwood, NJ, 1996, pp. 39-58.

Durso, F. T., and Gronlund, S. D. "Situation Awareness," in *Handbook of Applied Cognition,* F. T. Durso, R. S. Nickerson, R. W. Schvaneveldt, S. T. Dumais, D. S. Lindsay and M. T. Chi (eds.) John Wiley and Sons Ltd., New York, 1999.

Durso, F. T., Hackworth, C., Truitt, T. R., Crutchfield, J., Nikolic, D., and Manning, C. A. "Situation Awareness as a Predictor of Performance in En Route Air Traffic Controllers," *Air Traffic Control Quarterly* (6:1) 1998, pp. 1-20.

Green, T. R. G. "Conditional Program Statement and Their Comprehensibility to Professional Programmers" *Journal of Occupational Psychology* (50), 1977, pp. 93-109.

Halstead, M. *Elements of Software Science*, Elsevier North-Holland, New York, 1977.

Hatton, L. "Does OO Sync with How We Think?" *IEEE Software* (15:3), 1998, pp. 46-54.

Hendrix, D., Cross II, J. H., and Maghsoodloo, S. "The Effectiveness of Control Structure Diagrams in Source Code Comprehension Activities," *IEEE Transactions on Software Engineering* (28:5), 2002, pp. 463-478.

Hogg, D. N., Folleso, K., Strand-Volden, F., and Torralba, B. "Development of a Situation Awareness Measure to Evaluate Advanced Alarm Systems in Nuclear Power Plant Control Rooms," *Ergonomics* (11), 1995, pp. 394-413.

Jones, C. *Programmer Productivity*, McGraw-Hill, New York, 1986.

Jones, C. "Software Metrics: Good, Bad, and Missing," *IEEE Computer* (27:9), September 1994, pp. 98-100.

Kahneman, D. *Attention and Effort*, Prentice-Hall, Englewood Cliffs, NJ, 1973.

Kemerer, C. F. "Software Complexity and Software Maintenance: A Survey of Empirical Research," *Annals of Software Engineering* (1), August 1995, pp. 1-22.

Kemerer, C. F., and Slaughter, S. "An Empirical Approach to Studying Software Evolution," *IEEE Transaction on Software Engineering* (25:4), July/August 1999, pp. 493-509.

Koenemann, J, and Robertson, S. P. "Expert Problem Solving Strategies for Program Comprehension," in *Proceedings of CHI'89 Conference on Human Factors in Computing Systems*, S. P. Robertson, G. M. Olson, and J. S. Olson (eds.), ACM Press, New York, 1991, pp. 69-73.

Koch, I., and Prinz, W. "Process Interference and Code Overlap in Dual-Task Performance," *Journal of Experimental Psychology: Human Perception and Performance* (28:1), 2002, pp. 192-201.

Lientz, B. P., Swanson, E. B., and Tompkins, G. E. "Characteristics of Application Software Maintenance," *Communications of the ACM* (21:6), June 1978, pp. 461-471.

Littman, D. C., Pinto, J., Letovsky, S., and Soloway, E. "Mental Models and Software Maintenance," in *Empirical Studies of Programmers: First Workshop*, E. Soloway and S. Iyengar (eds.), Ablex Publishing, Norwood, NJ, 1986, pp. 80-98.

McCabe, T. J. "A Complexity Measure," *IEEE Transactions on Software Engineering* (SE-2:4), 1976, pp. 308-320.

Munson, J. C., and Khoshgoftaar, T. M. "The Dimensionality of Program Complexity," in *Proceedings of the 11th Annual International Conference on Software Engineering*, Pittsburgh, PA, 1989, pp. 245-253.

Navon, D. "Exploring Two Methods for Estimating Performance Tradeoff," *Bulletin of the Psychonomic Society* (28:2), 1990, pp. 155-157.

Navon, D., and Gopher, D. "On the Economy of the Human Processing Systems," *Psychological Review* (86), 1979, pp. 254-255.

Navon, D., and Miller, J. "Role of Outcome Conflict in Dual-Task Interference," *Journal of Experimental Psychology: Human Perception and Performance* (13:3), 1987, pp. 435-448.

Pashler, H. "Dual-Task Interference in Simple Tasks: Data and Theory," *Psychological Bulletin* (116:2) 1994, pp. 220-244.

Pashler, H., and O'Brien, S. "Dual-Task Interference and the Cerebral Hemispheres," *Journal of Experimental Psychology-Human Perception and Performance* (19:2), 1993, pp. 315-330.

Pennington, N. "Comprehension Strategies In Programming," in *Empirical Studies of Programmers: First Workshop*, G. M. Olson, S. Sheppard, and E. Soloway (eds.), Ablex Publishing, Norwood, NJ, 1987a, pp. 100-113.

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

Prechelt, L., Unger, B., Tichy, W. F., Brössler, P., and Votta, L. G. "A Controlled Experiment In Maintenance Comparing Design Patterns To Simpler Solutions," *IEEE Transactions on Software Engineering* (27:12), December 2001, pp. 1134-1144.

Prechelt, L., Unger-Lamprecht, B., Phillippsen, M., and Tichy, W. F. "Two Controlled Experiments Assessing the Usefulness of Design Pattern Documentation in Program Maintenance," *IEEE Transactions on Software Engineering* (28:6), June 2002 pp. 595-606.

Robson, D., Bennett, K. B. Cornelius, B., and Munro, M. "Approaches to Program Comprehension," *The Journal of Systems and Software* (14), 1991, pp. 79-84.

Sackman, H., Erickson, W. J., and Grant, E. E. "Exploratory Experimental Studies Comparing Online and Offline Programming Performance," *Communications of the ACM* (11:1), January 1968, pp. 3-11.

Sarno, K. J., and Wickens, C. D. "The Role of Multiple Resources in Predicting Time-Sharing Efficiency," *International Journal of Aviation Psychology* (5), 1995, pp. 107-130.

Shaft, T. M., and Vessey, I. "The Relevance of Application Domain Knowledge: The Case of Computer Program Comprehension," *Information Systems Research* (6:3), 1995, pp. 286-299.

Shaft, T. M., and Vessey, I. "The Relevance of Application Domain Knowledge: Characterizing the Computer Program Comprehension Process," *The Journal of Management Information Systems* (15:1), 1998, pp. 51-78.

Sharon, D. "Meeting the Challenge of Software Maintenance," *IEEE Software* (13:1), January 1996, pp. 122-125.

Sheppard, S. B., Curtis, B., Milliman, P., and Love, T. "Modern Coding Practices and Programmer Performance," *IEEE Computer* (12:12), 1979, pp. 41-49.

Sheil, B. A. "The Psychological Study of Programming," *ACM Computing Surveys* (13:1), 1981, pp. 101-120.

Shneiderman, B., Mayer R., McKay, D., and Heller, P. "Experimental Investigations of the Utility of Detailed Flowcharts in Programming," *Communications of the ACM* (20:6), 1977, pp. 373-381.

Sime, M. E., Green, T. R. G., and Guest, D. J. "Psychological Evaluations of Two Conditional Constructions Used in Computer

Languages" *International Journal of Man-Machine Studies* (5), 1973, pp. 105-113.

Stinchcombe, A. L. *Constructing Social Theories* Harcourt, Brace, and World, Inc., New York, 1968.

Tabachnick, B. G., and Fidell, L. S. *Using Multivariate Statistics* (4th ed.), Allyn and Bacon, Boston, 2000.

Taylor, R. M., Finnie, S., and Hoy, C. "Cognitive Rigidity: The Effects of Mission Planning and Automation on Cognitive Control in Dynamic Situations," paper presented at the Ninth International Symposium on Aviation Psychology, Columbus, OH, April 1997.

Vans, A. M., von Mayrhauser, A., and Somlo, G. "Program Understanding Behavior During Corrective Maintenance of Large-Scale Software," *International Journal of Human-Computer Studies* (51), 1999, pp. 31-70.

Van Selst, M., and Jolicoeur, P. "Decision and Response in Dual-Task Interference," *Cognitive Psychology* (33:3), August 1997, pp. 266-307.

Vessey, I. "Cognitive Fit: A Theory-Based Analysis of the Graph Versus Tables Literature," *Decision Sciences* (22:2), 1991, pp. 219-240.

von Mayrhauser, A., and Vans, A. M. "Industrial Experience with an Integrated Code Comprehension Model," *Software Engineering Journal* (10:5), September 1995, pp. 171-182.

von Mayrhauser, A., and Vans, A. M. "Identification of Dynamic Comprehension Processes During Large Scale Maintenance," *IEEE Transactions on Software Engineering* (22:6), June 1996, pp. 424-437

von Mayrhauser, A., Vans, A. M., and Howe, A. E. "Program Understanding Behavior During Enhancement of Large-Scale Software," *Software Maintenance: Research and Practice* (9), 1997, pp. 299-327.

Walz, D. B., Elam, J. J., and Curtis, B. "Inside a Software Design Team: Knowledge Acquisition, Sharing, and Integration," *Communications of the ACM* (36:10), 1993, pp. 63-77.

Weiser, M. "Programmers Use Slices When Debugging," *Communications of the ACM* (25:7), July 1982, pp. 446-452.

Whitaker, L. A. "Dual-Task Interference as a Function of Cognitive Processing Load" *Acta Psychologica* (43:1), January 1979, pp. 71-84.

Wickens, C. D. "Multiple Resources and Performance Prediction," *Theoretical Issues in Ergonomic Science* (3:2), 2002, pp. 159-177.

Wolfinger, R., and Chang, M. "Comparing the SAS GLM and MIXED Procedures for Repeated Measurements Analysis," *SAS Users Group International Proceedings*, Orlando, FL, April 2-5, 1995 (available online at http://support.sas.com/rnd/app/papers/mixedglm.pdf).

Zhang, J. "The Nature of External Representations in Problem Solving," *Cognitive Science* (21:2) 1997, pp. 179-217.

Zhang, J., and Norman, D. A. "Representations in Distributed Cognitive Tasks," *Cognitive Science* (57), 1994, pp. 87-122.

Zuse, H. *Software Complexity: Measure and Methods*, Walter de Gruyter, New York, 1991.

About the Authors

Teresa M. Shaft is an associate professor of Management Information Systems at The University of Oklahoma's Michael F. Price College of Business. She received her Ph.D. in Management Information Systems from the Pennsylvania State University. Her research interests focus on the cognitive processes of systems developers, the role of information systems in environmental management, and IT effectiveness. Her research appears in journals including *Information Systems Research, Journal of Management Information Systems, Database Advances*, and *Journal of Industrial Ecology*. Her research has been supported through grants from the U.S. National Science Foundation.

Iris Vessey is currently Honorary Professor at the University of Queensland and Adjunct Professor at the Queensland University of Technology. She received her M.Sc., MBA, and Ph.D. in Management Information Systems from the University of Queensland, Australia. Her research interests focus on the evaluation of emerging information technologies, knowledge management systems, and the management and organization of enterprise resource planning systems (ERPs). She serves, or has served, as an Associate Editor at *Information Systems Research*, *Journal of Database Management, Journal of Management Information Systems*, *MIS Quarterly,* and *Management Science*, and serves on the Executive Board of *Information Systems Frontiers*. During the first eight years of its life, she served as Secretary of the Association for Information Systems (AIS), as well as of the International Conference on Information Systems (ICIS) following its merger with AIS. She is an inaugural Fellow of the AIS.

Appendix A

Modification Specifications

Accounting Domain—Control Flow Task[8]

Currently the ROSTERFILE prints out subtotals for each department. Within each department, employees are listed in alphabetical order (on last name). Due to changes in reporting requirements, the roster format must be changed. Modify the program so that
(1) Subtotals are calculated for each budget code, within department.
(2) FICA withholdings are reported for each individual, each budget code, each department, and the university.

With the exception of the additions to the ROSTERFILE, the program outputs should remain the same.

A copy of the current ROSTERFILE and the required (i.e., modified) ROSTERFILLE are attached. The MASTERFILE has been reorganized to accommodate this change. The file definition of the MASTERFILE is not affected, just the ordering of the records. A copy of the new MASTERFILE is contained in the computer file BMAST.DAT. For testing purposes, copies of the necessary input files are provided on the computer.

The program is contained in EXAC.CBL. It contains the necessary changes to the record/file formats. The changes have been marked with a highlighter pen on the hard copy. A new BUDGET-LINE (which matches the format given in the modified ROSTERFILE) has been created. The necessary changes to the DEPT-LINE, UNIV-LINE, TITLELINE, and LINEOUT have also been made.

Please
(1) Implement the modification described above.
(2) Ensure that the modified program produces the requested outputs.

Hydrology Domain—Control Flow Task

Currently, the program prepares a REPORTFILE that lists statistics for seven parameters for each well. For each well, the statistics are printed out for each separate month. The city wants to keep a watch on each well, and wants the program modified to also print out the average and variance for each parameter for each well (i.e., for all months). Modify the program so that
(1) An overall average and variance are calculated for each separate parameter for each well, using all test values.
(2) The statistics are printed out on a separate page. Utilize the current page headers.

With the exception of the addition of the overall averages and variance, the program outputs should not change.

A copy of the current REPORTFILE and the required (i.e., modified) REPORTFILE are attached. For testing purposes, all necessary input files are provided on the computer.

The program is contained in EXHC.CBL. It contains the necessary changes to the record/file formats. The changes have been marked with a highlighter pen on the hard copy. The format for TOTAL-HEADER-RECORD has been added.

Please
(1) Implement the modification described above.
(2) Ensure that the modified program produces the requested outputs.

[8]Specifications were presented to participants with the title of "Modification." The application domain and modification task identifiers are included for explanatory purposes and were not provided to the participants.

Accounting Domain—Function Task

Employees often move and change their address. To simplify the process of updating addresses, we would like to include this capability in the program. An update code (ST-NEW-ADDRESS) has been added to the TIMECARD file. When the code is 'Y', the employee has a change of address. The new address is contained in the ADDRESS file. Modify the program so that

(1) The MASTERFILE is updated to contain the new address for those employees with a change of address.
(2) Other than changing the master file, the LABELFILE should use the new address.
(3) Should the social security number in the address file not match the masterfile social security number, write a message to the ERRORFILE.

With the exception of the updated MASTERFILE and the changes to the LABELFILE, the program outputs should not change.

A copy of the current MASTERFILE, the updated MASTERFILE, the old and new LABELFILE, the ADDRESS file, the new ERRORFILE, and the modified TIMECARD file are attached. For testing purposes, all necessary input files are provided on the computer.

The program is contained in EXAF.CBL. It contains the necessary changes to the record/file formats. The changes have been marked with a highlighter pen on the hard copy. The new ADDRESS file definition has been added. The ADDRESS-OUT record (to write an error message) has also been added.

Please

(1) Implement the modification described above.
(2) Ensure that the modified program produces the requested outputs.

Hydrology Domain—Function Task

New regulations have been passed concerning water quality. In an effort to comply with these regulations, limits for each of the seven parameters have been identified. When the average of a parameter is greater than this limit, the well must be rechecked by a hydrologist. Modify the program so that

(1) The AVERAGE is compared to the RECHECK value for each parameter.
(2) A message is written to the (new) RECHECKFILE when the AVERAGE is greater than the RECHECK limit. Each message should be placed on a separate page.

With the exception of the new RECHECKFILE, the program outputs should not change.

A copy of the required RECHECKFILE is attached. A copy of the new PARAMETERS-FILE is attached; it is also contained in the computer file PARAM2.DAT. For testing purposes, all necessary input files are provided on the computer.

The program is contained in EXHF.CBL. It contains the necessary changes to the record/file formats. The changes have been marked with a highlighter pen on the hard copy. The RECHECK field has been added to the PARAMETERS-FILE. New record formats to output the RECHECK message have been added (RECHECK-HEADER1, RECHECK-HEADER2, RECHECK-HEADER3, RECHECK-VALUES, RECHECK-LEVEL).

Please

(1) Implement the modification described above.
(2) Ensure that the modified program produces the requested outputs.

Appendix B

Rating Criteria for Modification Tasks

Accounting Control—Flow Modification Task

		Points Possible
1.	Recognize additional control break a. BUDGET level control break hold variable initialized in INIT (5) b. perform final BUDGET control break in WRAP-UP before CONTROL-BREAK (5) c. subjective: sees control break (5)	15
2.	Change SORT sequence a. add ST-BUDGET after LOCATION in SORT	10
3.	Place PERFORM of budget-level control break in the correct place a. always PRIOR to any CONTROL-BREAK (10) b. logic for BUDGET control break independent of CONTROL-BREAK (10)	20
4.	New paragraph for BUDGET control break	30
5.	Add new variables to Working Storage BUDGET-GROSS STUD-CT-BUDG BUDGET-NET BUDGET-WS	10
6.	Add FICA variables a. working storage (7.5) BUDGET-FICA DEPT-FICA UNIV-FICA b. change paragraphs to handle summing (7.5) DEPT — budget level and control break UNIV — control break and output final totals	15

Hydrology Control—Flow Modification Task

		Points Possible
1.	Recognize additional control break a. perform final FINISH-WELL in WRAP-UP after FINISH-MONTH b. subjective: sees control break (5)	15
2.	Place PERFORM of well level control break in the correct place after FINISH-MONTH in MATCH-WELL-IDS	20
3.	Update intermediate values in/after each FINISH-MONTH a. for new control break processing b. save NUM-OBS, SUM-X, SUM-X-SQUARED (10) c. separate for each parameter (10)	20
4.	Output WELL level stats for new control break processing a. separate page for the new statistics (6) b. each parameter is handled (6) c. compute stats properly (13)	25
5.	Add TABLE to hold WELL level values in working storage	20

Accounting Function Modification Task

	Points Possible
1. File handling	20
a. change MASTERFILE to open I/O (10)	
b. open (4)/close (3) ADDRESSFILE	
c. add EOF for ADDRESSFILE (3)	
2. Update ADDRESS paragraph	40
a. correct information (20)	
b. REWRITE MASTERFILE (20)	
3. Place/check address so LABELSFILE is properly updated	20
4. Error processing	20
a. recognize need for error processing (10)	
b. process for mismatched social security numbers (10)	

Hydrology Function Modification Task

	Points Possible
1. File handling (open/close RECHECKFILE)	20
2. Output RECHECK information paragraph	40
a. separate page/headers (10)	
b. correct information for each one (30)	
3. Check AVERAGE	20
a. for each parameter (10)	
b. correct place (10)	
4. Modify LOAD-PARAMETERS to input new variable	20