Communications of the Association for Information Systems

Volume 36 Article 4

2-2015

Task Mental Model and Software Developers' Performance: An Experimental Investigation

VenuGopal Balijepally
Oakland University, balijepa@oakland.edu

Sridhar Nerur
College of Business Administration, University of Texas at Arlington, TX

RadhaKanta Mahapatra
College of Business Administration, University of Texas at Arlington, TX

Communications of the Association for Information Systems

CAIS

Task Mental Model and Software Developers' Performance: An Experimental Investigation

VenuGopal Balijepally

School of Business Administration, Oakland University, MI

balijepa@oakland.edu

Sridhar Nerur

College of Business Administration, University of Texas at Arlington, TX

RadhaKanta Mahapatra

College of Business Administration, University of Texas at Arlington, TX

Abstract:

Our understanding of factors influencing the effectiveness of software-development processes has evolved in recent times. However, few research studies have furthered our understanding of the cognitive factors underlying software development activities and their impact on performance and affective outcomes. To some extent, this may be attributed to the paucity of measurement approaches available for cognitive factors. In this study, we fill this gap by developing a measurement approach to capture and evaluate the quality of mental models. We investigate the efficacy of mental models in software development using the said approach. We assessed mental model quality by statistically comparing a software developer's mental model with a referent model derived from multiple experts. Results of a controlled laboratory experiment suggest that a software developer's mental model quality is a determinant of software quality. Further, we found this effect to be consistent across software development tasks of varying complexities. These results not only shed light on the impact of mental models in software development, but also have significant implications for stimulating future research on cognitive factors influencing software development practices.

Keywords: Agile Methodologies; Mental Models; Pair Programming; Software Developer; Task Complexity.

Volume 36, Article 4, pp. 53-76, February 2015

The manuscript was received 13/06/2013 and was with the authors 6 months for 3 revisions.

I. INTRODUCTION

In today's business environment, organizations increasingly rely on software not only to streamline their processes, but also to gain and/or sustain their competitive advantage. Therefore, developing high software quality continues to be a top priority for organizations. However, achieving high software standards is not easy without understanding the cognitive challenges that confront software developers. Therefore, it's not surprising that some contemporary software practices, such as pair programming and test-driven development (Beck & Andres, 2005), were evolved to expressly address software's quality. While practitioners have been working to develop best practices, the acceptance and use of such practices are largely based on personal observations rather than on rigorous empirical validation (e.g., Zhang & Budgen, 2012). Software development research in recent times has focused on issues related to the efficacy of a programming pair compared to that of an individual programmer, test-driven development, and the diffusion of agile methods, but relatively little research has been devoted to elucidating the cognitive structures of pairs and individual developers in the software development context. Robillard (1999), for example, laments the lack of empirical research devoted to cognitive aspects of software development. Davern, Shaft, and Te'eni (2012), based on a reflective review of cognitive IS research, also note the paucity of research on the effect of current software development methods and practices on developers' cognitive processes, and highlight it as an enduring research question. They also urge IS researchers to explore the relationship between cognitive processes and emotions. Consistent with the software development research literature, we use the terms software developer, developer, and programmer interchangeably in the rest of the paper.

In the industrial and organizational psychology literature, where there is a long-standing tradition of research on individual and team cognition, both similarity (i.e., overlapping cognitions among team members) and accuracy of cognitive structures (referred to as schemas) have been shown to impact team effectiveness (Rentsch & Hall, 1994; Rentsch & Woehr, 2004). Early cognitive research in the domain of software development focuses on the processes that programmers use during software comprehension and categorized these into top-down (Brooks, 1983), bottom-up (Pennington, 1987; Shneiderman & Mayer, 1979), and opportunistic processes (Letovsky, 1987; Shaft & Vessey, 1998). With few exceptions (e.g., Shaft & Vessey, 2006), these studies conceive a developer's program comprehension, elicited mainly through predefined questions or free recall, as the dependent variable of interest. As such, they implicitly assume that a positive relationship between a developer's program comprehension and task performance exists (Shaft & Vessey, 2006). Shaft and Vessey (2006) suggest that the relationship between a software developer's comprehension and the quality of software modification is moderated by the cognitive fit between the task requirements and the knowledge emphasized in the developer's mental representation of the existing software. They operationalized cognitive fit and the knowledge emphasized in the maintenance task as experimental manipulations. Our study extends this stream of research by creating an approach to operationalize and measure a software developer's task mental model (which is similar to the mental representation of task solution articulated, but not directly measured, by Shaft and Vessey (2006)) and by explicitly studying its relationship with software quality in the context of individual and paired software development.

Mental models are internal representations of objects, people, situations, and actions. Kenneth Craik, a Scottish psychologist, pioneered this concept, while Johnson-Laird (1981) articulated the theory of mental models. According to Johnson-Laird (1980, p. 98), a mental model represents "a state of affairs and accordingly its structure…plays a direct representational or analogical role. Its structure mirrors the relevant aspects of the corresponding state of affairs in the world.". Individuals construct mental models based on experience and observation of a particular entity of interest or of the world in general (Wilson, 2000). Indeed, a mental model that is structurally compatible with a domain may be generated by a semantic analysis of verbal statements pertaining to the domain (Johnson-Laird, 1983). As per cognitive learning theory, analogies and metaphors help individuals to quickly construct an initial mental model of a new domain by facilitating the mapping of concepts and interrelationships from a known domain to the new one. For instance, analogies help people build a structural map that simulates the way a system's components interact (Collins & Gentner, 1987). Depending on the context, individuals may develop and use several different mental models.

Enhancing software developers' effectiveness and ensuring their software projects succeed through new methodologies, tools, and practices is of continuing interest to the software community. Understanding the effect of different methods and practices on developers' cognitive functions and processes is an enduring research question (Davern et al., 2012) that could help establish the efficacy of these methods and improve future practice. As Davern et al. (2012) emphasize, exploring the interconnection between cognition and emotion provides potentially richer

explanations for the drivers of human behavior. To this end, with this research, we shed light on the possible effects of software developers' task mental models on their performance and affective outcomes in different programming settings. Specifically, we explore the following research questions:

1. When working on a software maintenance task, does the quality of a software developer's task mental model affect that individual's task performance and affective perceptions?

2. Does the programming setting (individual vs. paired development) or a task's complexity (low vs. high) have any effect on the relationship between the quality of a software developer's mental model and task performance?

The paper is organized as follows. In Section 2, we sample the literature on mental models and derive research hypotheses for the programming context. In Section 3, we explain the research model and hypotheses. In Section 4, we present the research method and, in Section 5, we discuss the analysis, results, and implications. Finally, in Section 6, we conclude the paper by noting the importance of this study for future investigations into the cognitive aspects of software development.

II. MENTAL MODEL THEORY

Researchers have found the mental model concept to be useful across multiple fields. In cognitive psychology, where the concept originated, mental models are used to explain mental processes. In applied fields such as software development and human factors, mental models help capture the outcomes of mental processes. In the systems dynamics literature, a mental model of a dynamic system is defined as "a relatively enduring and accessible, but limited, internal conceptual representation of an external system (historical, existing or projected), whose structure is analogous to the perceived structure of that system" (Doyle & Ford, 1999, p. 414). In the human factors literature, mental model refers to "the user's mental representation of the components and operating rules of the system…[that] may vary with respect to its completeness and veridicality" (Cañas, Bajo, & Gonzalvo, 1994, p. 795). In early systems development research, mental model denoted a developer's knowledge about a system (Littman, Pinto, Letobsky, & Soloway, 1987).

The mental model concept has its theoretical roots in functionalism, a philosophical approach that allows one to define mental states in relation to their causal effect on other mental states or behaviors (Stubbart, 1989). Mental model theory competes with the premise that deductive reasoning in human mind is driven by formal rules of inference. Instead, it argues that much of human cognition involves creating and manipulating mental models. Unlike traditional psychological theories where formal rules of logic help refute or validate deductive inferences, the validity of a mental model-based inference is tested by searching for alternative models that refute it (Johnson-Laird, 1995).

Mental models help individuals comprehend a phenomenon of interest and make inferences and predictions related to its state and/or behaviors. Thus, mental models enable individuals to experience events and situations by proxy and help them make decisions to adequately handle tasks (Johnson-Laird, 1983). Depending on the phenomenon, the mental models that individuals construct may vary in their levels of abstraction (Wilson & Rutherford, 1989). Mental models help organize knowledge in robust, parsimonious ways and reduce complexity. Thus, they enable one to process information efficiently by making it unnecessary for the individual to understand from scratch each time a novel situation is encountered. They direct the perception and processing of stimuli, which, in turn, help shape or change mental models (Vandenbosch & Higgins, 1996).

Prior research in the information systems (IS) domain has demonstrated that executive support systems (ESS) help users preserve the mental model of a particular domain through focused search. ESS were also found to assist executive users with developing mental models when they engaged in solving problems that were not clearly formulated (Vandenbosch & Higgins, 1996). IS training literature attests to performance benefits for subjects who develop conceptual mental models during training over those who develop procedural mental models (Santhanam & Sein, 1994).

A limitation of the mental model concept is that its measurement is closely tied to the experimental paradigm and may not be a true translation of the internal representation of the mental model. However, researchers consider the mental model to be a "useful heuristic" for exploring individual/team cognition because it encompasses both knowledge and belief structures (Langan-Fox, Anglim, & Wilson, 2004). This paper focuses on individual mental models and empirically affirms the usefulness of measuring mental models in the context of software programming.

III. RESEARCH HYPOTHESES

In the context of solving word problems, Stern (1993) articulates two situational models that help problem solvers understand and represent the problem at hand—the episodic situation model (Reusser, 1990) and the problem model (Riley & Greeno, 1988). The episodic situation model enables one to understand the context of a specific word problem. The problem model, on the other hand, includes information—both structural and relational—that is germane to such a problem. Thus, it helps in evolving an appropriate mathematical model for solving a word problem (Stem, 1993). In the program comprehension literature, a software developer's mental representation of the entities of the problem domain and their relationships is called a situation model (Burkhardt, Détienne, & Wiedenbeck, 2002, Pennington, 1987).

Drawing on the problem model in the word-problem literature (Riley & Greeno, 1988) and the situation model in the program-comprehension literature (Pennington, 1987), we conceptualize the task mental model (TMM) to include both conceptual and relational information relevant to a programming task. TMM represents a software developer's understanding of the relationships among various objects and behaviors (methods) associated with a task. We argue that a software developer's TMM drives their search for an appropriate programming solution, which ultimately influences the software solution's quality. While software quality relates to the implemented solution, based on the particular language's semantics and syntax, task mental model represents the instantiated knowledge structures (Wilson & Rutherford, 1989) facilitating such a solution. Our research questions focus on the effect of a software developer's task mental model on software quality and the moderating effect of task complexity and programming setting (individual vs. paired development) on this relationship. In addition, we also examine the effect of TMM on the software developer's affective responses, such as task satisfaction and confidence in performance.

Task Mental Model and Software Quality

A software maintenance task requires a software developer to comprehend the components of the system and their interrelationships, and to integrate information from multiple domains to code the software artifact that satisfies the system specifications. To accomplish this, software developers cognitively build and refine a TMM of the system that reflects their current understanding of the system components and their interrelationships. Software developers iteratively refine their initial TMM, which is based on their early understanding of the system, as more information becomes available during the course of the system-development process.

The extant research on program comprehension has identified three distinct strategies that developers use during a comprehension task. These are the top-down (Brooks, 1983), bottom-up (Pennington, 1987; Shneiderman & Mayer, 1979), and opportunistic strategies (Letovsky, 1987; Shaft & Vessey, 1998). The top-down strategy, typically used in more familiar problem domains and programming language environments (Shaft & Vessey, 1995; Soloway & Ehrlich, 1984), involves building knowledge first at the level of the problem domain and translating it into the source code (Brooks, 1983). The bottom-up strategy involves reading the code and cognitively grouping lines of code to build higher level abstractions. By repeating this process multiple times, the software developer progressively develops a higher level of understanding of the program (Shneiderman & Mayer, 1979). Letovsky (1987) describes programmers as opportunistic processors who use their knowledge base (i.e., knowledge relating to application domain, programming domain, and problem-solving approaches) to evolve mental models through an assimilation process, which may be top-down or bottom-up depending on their initial knowledge base. Shaft and Vessey (1998) refer to this as a flexible comprehension process. Program comprehension research also offers some anecdotal evidence to suggest that superior understanding contributes to successful program enhancements (Littman et al., 1987).

In a study (Shih & Alessi, 1993) concerning code evaluation in programming, conceptual models helped improve programmers' conceptual understanding of the programming task as reflected in their mental models. This study also found the quality of the mental models to be positively related to the transferability of procedural skills from code evaluation to code generation. In a different context (namely, electronic troubleshooting), Rowe and Cooke (1995) argue that the quality of an individual's mental model is positively associated with performance.

We use an expert' software developer's TMM for a particular software task as the benchmark when measuring the quality of a software developer's mental model for that task. Thus, a statistically significant correlation between a software developer's TMM and that of an expert software developer (referred hereafter to as just expert, in the interest of brevity) would signify a developer's superior understanding of the system. From the theoretical perspective of human problem solving (Newell & Simon, 1972), understanding a problem space's semantics (i.e., its concepts and relationships) can help structure the problem and facilitate its solution. Prior empirical findings suggest that higher software comprehension leads to superior performance in software maintenance tasks only when there is a cognitive fit between the requirements of a maintenance task and the knowledge emphasized in a software developer's mental representation (Shaft & Vessey, 2006). We expect an individual developer to achieve higher

software quality when the individual's TMM is significantly correlated to that of an expert than when it is not. We also expect this relationship to hold in a paired programming setting (see following section). We consider a pair's TMM to be "superior" if the TMM of either member is significantly correlated to that of an expert (see following section).

In a paired development setting, two developers collaboratively code software using XP (extreme programming) procedures. The programming task is shared: one developer codes at the keyboard, while the other inspects the code and helps think strategically about the programing task. The partners switch their roles at regular intervals (Beck & Andres, 2005; Williams & Kessler, 2000). Drawing from small group research, we can categorize programming tasks as intellective tasks (i.e., tasks with demonstrably correct solutions) (Laughlin, 1980). They may also be considered disjunctive tasks (i.e., the group performance is determined by group member with the best solution) according to Steiner's (1972) task typology. Thus, in pair programming, pair performance is largely determined by the member with the superior solution to the task at hand. That is, the group successfully solves a problem when any one group member can figure out the solution to the problem. However, we argue that the member with the superior TMM would drive the pair solution. Thus, we expect the software quality achieved on a maintenance task by a programming pair using XP procedures to be higher when it has a superior TMM (i.e., the TMM of any of its members is significantly correlated to that of an expert). As such, we hypothesize:

> **H1.** *When working on a software maintenance task, an individual software developer or a collaborating pair of software developers with a superior TMM will achieve higher performance—measured in terms of software quality—compared to an individual/pair without a superior TMM.*

Task Mental Model and Developer Satisfaction

As we mention earlier, superior mental models, reflecting enhanced comprehension of a task, provide cues and positively guide problem-solving behavior. Such cues help reduce a software developer's cognitive burden during problem solving (e.g., Storey, Fracchia, & Muller, 1997). During task performance, software developers gain a better understanding of the problem at hand as they continually examine the results of their programming efforts. That is, a greater sense of one's level of program comprehension unfolds as one codes, compiles, and debugs the program. On the contrary, when a software developer struggles with a programming task and is unable to reach a good understanding of the problem domain, the individual again would be able to sense this and feel frustrated. The effect of an individual's cognition on their emotions is well documented in the research literature. Negative cognitions among ICT (information and communication technology) users due to information overload and the demands of computer usage are known to cause "technostress" and lower individuals' satisfaction with ICT systems (Tarafdar, Tu, & Ragu-Nathan, 2010). Thus, it is reasonable to expect that a software developer who reaches a good understanding of a problem is likely to feel more satisfied with task performance than one who is unable to attain such an understanding. Specifically, we expect a software developer, when working individually, to be more satisfied with task performance when their TMM is more closely aligned with that of the expert than when it is not.

In a programming pair, when either member is able to achieve a superior understanding of the problem domain, the pair certainly becomes aware of it based on how they are able to code, test, and make the program they are working on perform adequately. Because the person with the better understanding of the problem domain could demonstrate the resulting solution to the partner, any positive affect experienced by one member would quickly spread to the other due to their close interaction. In contrast, when neither member has a superior understanding, the resulting frustration would also be shared among both members. Therefore, we expect mean satisfaction to be higher among a collaborating pair when either of its members has a TMM closely aligned with that of the expert than when it is not. As such, we hypothesize:

> **H2.** *When working on a software maintenance task, an individual software developer or a collaborating pair of software developers with a superior TMM will experience higher satisfaction compared to an individual/pair without a superior TMM.*

Task Mental Model and Developer Confidence

Cognitive psychology research suggests that, in general, people can successfully monitor and evaluate their memories (Koriat, Lichtenstein, & Fischhoff, 1980), although some systematic distortions could occur (McKenzie, 1997). According to this research, individuals judge their confidence in solving a task based on the task structure and the structure of the known environment in their long-term memory (Gigerenzer, Hoffrage, & Kleinbölting, 1991). Individuals base their confidence in a solution on the strength of evidence retrieved for that solution relative to alternative solutions (Griffin & Tversky, 1992). Thus, requiring people to consciously list and weigh the evidence—both supporting and disconfirming—when evaluating a solution improves their calibration of their confidence judgments (Koriat et al., 1980).

In a programming task, we expect software developers to base their confidence judgments on the evidence retrieved from their TMMs. When software developers perform tasks involving iterative cycles of coding, compiling, and debugging, we expect them to continually weigh the evidence and counter-evidence of various alternatives/solutions before creating their final solutions. Accordingly, we expect their confidence judgments to be closely aligned to how well they understand the problem's domain. Specifically, we expect software developers, when working individually, to have higher confidence in their solutions when their TMMs are more closely correlated with the expert mental model than when they are not.

In a paired development setting, we expect the software developer with the superior TMM to base their confidence judgments about the solution on the evidence and counter-evidence retrieved from their TMM. Due to collaboration and constant communication inherent in a pair programming setting and due to the intrinsically high solution demonstrability of programming tasks, we expect the second software developer also to calibrate their confidence judgments in light of the evidence for the effective solution. Accordingly, we expect a pair's mean confidence in their performance to be higher when either member has a superior understanding of the problem domain compared to when they don't. As such, we hypothesize:

> **H3.** *When working on a software maintenance task, an individual software developer or a collaborating pair of software developers with a superior TMM has higher confidence in performance compared to an individual/pair without a superior TMM.*

Moderating Effects of Task Complexity and Programming Setting

Task complexity enhances cognitive demands on a software developer through an increase in information diversity, rate of information change, and information load (Campbell, 1988). Task complexity typically increases solution ambiguity, which makes the end state less obvious. Software developer pairs may also experience process ambiguity in structuring an effective collaborative process for working toward a programming solution (Helquist, Deokar, Meservy, & Kruse, 2011). Thus, when performing a more complex programming task—one that has intrinsically higher solution ambiguity and higher performance risk relative to a less-complex task—it is reasonable to expect that achieving a superior understanding of the problem domain would result in higher performance rewards. In the absence of any prior empirical evidence to the contrary, we expect increased software quality for developers (both individual developers and pairs) having superior TMM when task complexity is high than when it is low. As such, we hypothesize:

> **H4.** *Task complexity will accentuate the software quality benefits for a software developer or a programming pair of software developers with a superior TMM.*

Proponents of XP ascribe several benefits to software developers in a paired development setting over software developers in an individual development setting—enhanced learning (Williams, 2000), higher software quality (Nosek, 1998; Williams & Kessler, 2000), and greater satisfaction and confidence in the solution (Williams, 2000). However, the empirical evidence relating to the software quality benefits of paired development is at best mixed (Arisholm, Gallis, Dyba, & Sjoberg, 2007; Nawrocki & Wojciechowski, 2001). The distributed cognition theory (Flor & Hutchins, 1991) ascribes information processing benefits to pairs over individuals; for example, the ability to search through a larger space of alternatives, ready access to shared memory of old plans, and ability to jointly develop ambiguous code segments with fewer defects. Active communication between partners could facilitate perspective taking and perspective making (Boland & Tenkasi, 1995), which can particularly benefit the member who can better comprehend a particular task and code the solution.

In programming dyads, there are potential process losses that could affect pair performance. The group literature alludes to dysfunction and motivational losses inherent in group work (e.g., social loafing and social facilitation). Social loafing occurs when individuals exert less effort when working in groups than they would when working individually (Karau & Williams, 1993). Feeling of reduced responsibility for group performance (Petty, Harkins, Williams, & Latane, 1977) and the perception that an individual's effort is not identifiable (Williams, Harkins, Latane, 1981) or that one's effort is dispensable (Harkins & Petty, 1982) are some factors that promote social loafing. Social facilitation occurs when a person works in the presence of an observer: the observer facilitates the individual's performance on well-learned tasks, but hampers their performance on novel or difficult tasks. The elevated drive levels due to the feeling of being evaluated, or cognitive distraction due to the presence of the observer are some possible causes for this effect (Aiello & Douthitt, 2001; Zajonc, 1965).

Thus, both motivational losses and information processing benefits are distinct possibilities in a paired programming setting. However, consistent with prior agile literature, we hypothesize that having superior understanding of the problem domain will yield higher software quality in a paired development setting over an individual setting. As such, we hypothesize:

H5. *A paired software development setting, compared to an individual software development setting, will accentuate software quality benefits for software developers with superior TMMs.*

IV. METHOD

Our research is part of a larger experimental study conducted to investigate the effectiveness of paired versus individual programming. The larger study involved a laboratory experiment using a 2 x 2 factorial design. The experimental design involved manipulating two main factors: programming setting (individual versus paired development) and task complexity (low versus high). In the individual condition, individual participants worked on a programming task; in the paired condition, two participants worked together on the task using XP procedures. Research findings from the larger study relating to a different research question appeared in Balijepally, Mahapatra, Nerur, and Price (2009).

Our current research examines the effect of task mental models on software developers' performance, which differs substantially from what was reported in Balijepally et al. (2009). Accordingly, it involves new data relating to developers' mental models and a fresh analysis to test the hypotheses associated with the research questions we address here. The mental model data that underpins our current study was not used in Balijepally et al. (2009).

Participants

The experiment involved student subjects as surrogates for entry-level software developers. We recruited the subjects from among students enrolled in undergraduate and graduate IS courses in a large public university in the USA. Participation in the study was voluntary and students received class credit for their participation. Knowledge of Java programming was a prerequisite for participation. We conducted the experiments over three semesters. We randomly assigned the 122 student subjects who signed up for the study to the individual or the paired condition. We dropped data related to five subjects assigned to the individual condition due to incomplete information provided on the background and/or mental model questionnaire. This resulted in a final subject pool of 57 in the individual condition and 60 in the pair condition. We again randomly assigned the subjects in the individual and the pair conditions to work on one of the experimental tasks: a low-complexity task or a high-complexity task. The demographic details of the subjects were as follows: 96 undergraduates, 21 graduates; 86 men, 31 women. In addition, subjects had an average programming experience of 1.92 years.

Experimental Task

We varied task complexity across the two levels (low and high) by designing tasks varying in multiplicity of solution paths (Campbell, 1988). This paper's second author designed the two experimental tasks (i.e., a low-complexity task and a high-complexity task) and a warm-up task used in the experiments. The low-complexity task required subjects to modify five methods in two classes, while the high-complexity task required subjects to modify seven methods in five classes.

Experimental Procedure

The experiments occurred in the aforementioned university's college of business's research lab. We first conducted a pilot study to test the experimental protocols for each treatment condition. We used the feedback the participants provided about the experimental task, time duration, lab setting, and programming environment to fine-tune the scripts, session durations, and experiment logistics. Subjects worked on the assigned experimental tasks in insulated cubicles equipped with laptop computers. The computers were loaded with Java SDK5 and related API documentation, but were not connected to the Internet. We provided subjects in the pair condition with written instructions on working collaboratively as per XP procedures. Subjects worked on a warm-up task for the first 15 minutes to familiarize themselves with the computing platform and the lab setting. In addition, the warm-up provided subjects in the pair condition with the opportunity to familiarize themselves with each other and with the procedures for collaborative work.

Following these steps, the participants performed the experimental task with a maximum duration of two hours. The main experimental tasks were program maintenance tasks. We provided subjects with partial code and they had to augment it with new code to meet the program specification. As a manipulation check and to confirm that students in the pair condition took turns at the keyboard, the experimenter reminded them to do so every fifteen minutes. The experimenter also visited the subjects in the individual condition every thirty minutes to ensure that they adhered to the experimental protocol.

Elicitation and Evaluation of Task Mental Model

Two academics, including the second author, with considerable experience in teaching Java served as the software development experts. Neither was involved in running the experiments. Both experts jointly identified the most

important classes and methods associated with the two experimental tasks. A pair-wise comparison of these concepts formed the basis of eliciting subjects' mental models. Our underlying premise was that the perceived strengths of relationships among these concepts, identified by a subject immediately after completing the programming task, reflected the individual's understanding of the task. Similar techniques have been widely used in various research domains, including cognitive psychology (Durso & Coggins, 1990), team mental models (Edwards, Bell, Day, & Arthur, 2006), and software requirement understanding (Kudikyala & Vaughn, 2005).

At the end of each experimental session, each participant filled out a survey that included demographic details and questions related to the concepts of interest in the domain. Specifically, subjects had to rate the perceived strengths of the relationships among salient concepts identified by the experts for the experimental task they performed using a Likert scale ranging from (1) not at all related to (7) highly related (see Appendices A and B). The low-complexity task involved seven concepts, whereas the high-complexity task included 10 concepts.

Each subject's response to the TMM questions resulted in a symmetrical matrix, with diagonal values entered as 7. The size of such a matrix depended on the complexity of the experimental task, with the low-complexity task resulting in a 7 x 7 matrix and the high-complexity task producing a 10 x 10 matrix. In the paired-programming condition, subjects provided these details individually so that their individual TMMs could be constructed. The two experts jointly developed the expert TMM for each experimental task, which served as benchmarks for evaluation. We considered a subject's TMM to be superior if it had a significant correlation with the expert TMM (i.e., $p <= 0.05$). For the pairs, if either member had a superior TMM, then we considered the pair to have a superior TMM.

Dependent Variables

The dependent variables were software quality, task satisfaction, and confidence in performance. We measured software quality by assessing the quality of the programming solutions that the subjects developed. Two doctoral students not directly connected to the study independently scored the solutions' quality using a common assessment rubric designed for the experimental tasks. They evaluated the solutions on a scale of 0 to 125. In five cases, there were major differences between the scores assigned by the two raters, but they resolved these differences through discussions. The resulting software quality scores assigned by the two raters were highly correlated (Pearson Correlation = 0.983). The average of the raters' scores constituted the measure of software quality for each solution.

The second dependent measure of satisfaction represented each subject's affective response to the overall experience of completing the programming task. We adapted the measure developed by Bhattacharjee (2001) to assess satisfaction. It required participants to respond to the question "How do you feel about your overall experience of working on the programming task today?" using a Likert scale ranging from: (a) 1—very dissatisfied to 7—very satisfied; (b) 1—very displeased to 7—very pleased; (c) 1—very frustrated to 7—very contented; and (d) 1—absolutely terrible to 7—absolutely delighted. The mean score of these items served as the measure of satisfaction (Cronbach's $\alpha = 0.934$).

The third dependent measure, confidence in performance, denoted the strength of a subject's belief concerning the quality of their programming solution. We adapted the measure for this variable (Cronbach's $\alpha = 0.945$) from the existing literature (Brewer & Kramer, 1986; Jourden & Heath, 1996). Unlike software quality, which represents a subject's objective performance in a programming task, we designed confidence in performance to capture a subject's broad perception of their performance. Subjects responded to the question, "How do you feel about the quality of your programming solution?" using a Likert scale ranging from: (a) 1—not at all confident to 7—very confident and (b) 1—not at all certain to 7—very certain. Subjects also responded to the question "Imagine that we selected ten results at random from those who participated in this task. How would your performance rank among these ten results?". They ranked their performance using a Likert scale ranging from: 1—worst result out of ten to 10—best result out of ten. Because one item measured confidence in performance on a 10-point scale and two items on a 7-point scale, we created a summated scale for the measure (i.e., by first adding standardized individual items and then standardizing the resultant summated variable). The mean and standard deviation of the resultant measure of confidence in performance were 0 and 1, respectively.

We recognize that a developer's programming ability affects their performance (i.e., software quality achieved). We distinguish programming ability from other constructs as follows. While programming ability represents the skills a developer brings to bear on a task based on their prior training and experience, task mental model represents the developer's understanding of the specific software task's components. Software quality, on the other hand, captures the extent to which the software solution meets the expected features and specification.

We measured programming ability using a weighted average GPA of the grades the subject earned in all IS courses that the individual had taken. We weighted the grades earned in programming and systems analysis and design courses twice as much as those earned in other IS courses. We used this modified GPA (hereafter referred to as

GPA, for brevity) as a covariate in the statistical analysis to control for variability in the software quality scores stemming from differences in participants' programming abilities.

To help check the success of task complexity manipulation, we used a two-item measure of perceived task complexity that required participants to respond to the question "How do you feel about the main programming task, as compared to the warm-up task?" on Likert scales varying from (1) very easy to (2) very difficult and (1) very simple to (2) very complex. We used the mean scores across the two items to denote the perceived task complexity (Cronbach's $\alpha = 0.87$). To check the success of the task complexity manipulation, we then conducted a 2 (TMM quality—superior vs. inferior) x 2 (task complexity—low vs. high) ANOVA analysis on the dependent measure of perceived task complexity. Results suggested a significant main effect for the task complexity manipulation ($F = 21.136$, $p < 0.001$) on the perceived task complexity measure. Thus, the participants in the high-complexity task condition perceived the task to be more complex ($M = 5.239$) than those in the low-complexity task condition ($M = 4.167$), which provided assurance regarding the success of the task complexity manipulation. In addition, we obtained independent evaluations from two experts on the perceived complexity of tasks using the same two-item measure used above. The two experts had several years' experience teaching Java programming but were not connected in any way with the experiments or the study. Based on the mean scores for the two items, the experts rated the task complexity of the two tasks to be very different ($M = 5.75$ and $M = 3.75$ for high-complexity and low-complexity tasks, respectively) compared to the warm-up task, which provided further assurance on the success of the experimental manipulation.

V. ANALYSIS AND RESULTS

Because each collaborating pair created a single solution, we measured the dependent measure of software quality as a group-level construct for the paired-programming condition. For the individual programmers, we measured software quality individually. We measured the other two perceptual dependent measures—satisfaction and confidence in performance—individually for each member of the collaborating pair and then averaged the results for each pair.

We used UCINET's quadratic assignment procedure (QAP), a technique that relies on permutations (see Hubert, 1985; Krackhardt & Porter, 1986; Prell, 2012), to test for significance of the association between the developers' TMM and the expert TMM. As we describe earlier, each TMM is a symmetric matrix of relationships between concepts drawn from the domain. That is, each cell in the matrix contains dyadic information about the level of perceived similarity between two concepts in the problem space. These two matrices—an independent matrix (say, IV matrix) and a dependent matrix (say, DV matrix)—serve as inputs to the QAP procedure. QAP then determines the significance of the correlation between the two matrices in the following manner (Prell, 2012; Simpson, 2001):

1. It calculates the correlation coefficient (say, x) between the IV and DV matrices.

2. It then randomly permutes the rows and columns of the DV matrix in such a manner that the elements in a row/column will be the same as the original DV matrix, albeit in a different order. This ensures that the values associated with a concept, in a row and in a column, are not changed. The correlation coefficient (say, y) between the IV matrix and permuted DV matrix is then recalculated. This process is repeated thousands of times to obtain a reference sampling distribution.

3. A comparison of the correlation between the original matrices (i.e., x) and the generated sampling distribution indicates whether the observed correlation is a chance occurrence or if it indeed represents a significant level of similarity between the two matrices.

Researchers have used various techniques, including multidimensional scaling (MDS) (Rentsch & Klimoski, 2001), pathfinder's closeness metric (C) (e.g., Edwards et al., 2006), and UCINET's quadratic assignment procedure (QAP) (e.g., Mathieu, Heffner, Goodwin, Salas, & Cannon-Bowers, 2000) to assess similarity between two mental models. Since we wanted to determine the statistical significance of the correlation of each subject's TMM with the expert TMM, we deemed QAP to be more appropriate. In addition, Krackhardt and Porter (1986) list several advantages of this procedure. First, unlike linear models, QAP allows researchers to compare two matrices for similarity. Second, it is a nonparametric technique that is not sensitive to departure from the assumption of independence of dyads. Third, by comparing the corresponding cells of the two matrices, it "takes advantage of all the dyadic information represented in each matrix" (Krackhardt & Porter, 1986, p. 52). This is in contrast to Pathfinder, which uses the overlap in links between two networks (graphs) as the basis for computing the closeness metric.

For this analysis, we coded a subject's TMM (expressed as a matrix of the strength of relationship ratings between salient concepts of the problem domain) as superior if it was significantly correlated ($p < 0.05$) to the expert TMM for that task. If not, it was coded as inferior.

We first conducted a preliminary 2 x 2 x 2 MANCOVA analysis to assess the overall significance of the impact of the three independent factors across the set of dependent variables (software quality, satisfaction, and confidence in performance). Results suggested that TMM quality (superior group x inferior group) had a highly significant effect on the set of dependent measures (Wilks' Lambda = 0.813, $F = 5.845$, $p < 0.01$), as did task complexity (Wilks' Lambda = 0.819, $F = 5.611$, $p < 0.01$) and programming ability (Wilks' Lambda = 0.764, $F = 7.804$, $p < 0.01$). However, we did not find programming setting (individual developers x paired developers) to be significant (Wilks' Lambda = 0.947, $F = 1.407$, $p = 0.247$), nor did we find interactions of TMM quality x programming setting (Wilks' Lambda = 0.976, $F = 0.613$, $p = 0.609$), TMM quality x task complexity (Wilks' Lambda = 0.980, $F = 0.524$, $p = 0.667$), and TMM quality x programming setting x task complexity (Wilks' Lambda = 0.993, $F = 0.171$, $p = 0.916$) to be significant.

However, based on the preliminary analysis, we realized that several observations in certain groups were inadequate for a rigorous 2 x 2 x 2 MANCOVA analysis. Because TMM quality was not based on direct experimental manipulation, we could not control the sample size in the different groups as part of the experimental design. Therefore, based on pragmatic considerations, we decided to carry out only a 2 x 2 MANCOVA using the following two factors: TMM quality and task complexity. Because we did not find programming setting and its interactions to be significant in the preliminary analysis, we dropped programming setting from further analysis and accordingly grouped the data on this dimension. We realize that this could increase the variance and could reduce the effect size for the other two factors. However, we took comfort in the fact that this would be a more conservative approach—in terms of the findings of the statistical analysis—that avoids a false positive (type I error), but errs on the side of a false negative (type II error). This approach also helped us to rigorously test the underlying assumptions of the MANCOVA analysis and thus increase our confidence in the findings.

Assumption Checks

We first checked if the dependent measures satisfied the assumptions for applying a 2 x 2 MANCOVA procedure. The MANCOVA requirement for the presence of significant correlations among the dependent measures was satisfied based on Bartlett's test for sphericity (Chi-Square = 799.818, $p < 0.01$). The linearity assumption was satisfied because we found no non-linear relationships in the scatter plots matrix of the dependent measures and the covariate. Based on Shapiro Wilks tests, the normality assumptions were satisfied for the three dependent measures in each factorial group. The assumption of equal variance across treatment groups was satisfied based on the modified Levine test for software quality ($F = 0.885$, $p = 0.452$), satisfaction ($F = 0.383$, $p = 0.766$), and confidence in performance ($F = 2.394$, $p = 0.074$). Note that F test in ANOVA models is robust against violations of normality (Neter, Kutner, & Nachtsheim, 1996). Box's M test for the assumption of the equality of variance-covariance matrices across groups was, however, significant at the 5 percent level ($M = 32.921$, $p = 0.035$). Because Box's M test is considered to be very sensitive to any violations of normality, when the test results are significant, one suggested approach is to do MANOVA significance testing at a more conservative level (say, 3%) (Hair, Black, Babin, Anderson, & Tatham, 2006). Accordingly, we followed this approach in judging the statistical significance of the MANCOVA results.

Internal Validity Checks

Because we conducted the experimental sessions over three semesters, we conducted separate one-way ANOVA analyses to check for any systematic biases. The tests revealed no significant differences across semesters among the dependent measures of software quality ($F(2, 87) = 0.54$, $p = 0.58$), satisfaction ($F(2, 87) = 0.46$, $p = 0.63$), and confidence in performance ($F(2, 87) = 1.01$, $p = 0.37$), which provides assurance against any time-ordered effects.

MANCOVA Results

Based on satisfactory results on the assumption checks, we proceeded to conduct a 2 x 2 MANCOVA procedure to check the impact of the two factors (TMM quality and task complexity) across the set of dependent variables (software quality, satisfaction, and confidence in performance). Doing MANCOVA analysis before doing ANCOVA analyses for each dependent measure is expected to help guard against inflated type I error (Hair et al., 2006). The results of the 2 x 2 MANCOVA procedure suggested that TMM quality (superior group x inferior group) had a highly significant effect on the set of dependent measures (Wilks' Lambda = 0.792, $F = 7.012$, $p < 0.01$), as did task complexity (Wilks' Lambda = 0.826, $F = 5.602$, $p < 0.01$) and programming ability covariate (Wilks' Lambda = 0.776, $F = 7.698$, $p < 0.01$). However, the interaction of TMM quality x task complexity (Wilks' Lambda = 0.990, $F = 0.271$, $p = 0.846$) was not significant.

Given the significance of the MANCOVA model for the two independent factors and the programming ability covariate, we conducted 2 x 2 ANCOVA analyses for each of the dependent measures with programming ability used as a covariate in each analysis. Table 1 summarizes the ANCOVA results. Table 2 shows the means and standard deviations of the three dependent measures for the two independent factors. Figure 1 illustrates the plots of the marginal means for the three dependent measures.

Table 1: One-Way ANCOVA Results for the Dependent Measures

	SS	df	MS	F	p-value
Software quality (SQ)					
Task mental model quality (TMM)	9278.965	1	9278.965	15.330	0.000*
Task complexity (TC)	9450.486	1	9450.486	15.613	0.000*
Programming ability (GPA)	14205.568	1	14205.568	23.469	0.000*
TMM x TC	178.261	1	178.261	0.295	0.589
Error	49633.106	82	605.282		
Total	383527.750	87			
Model R squared = 0.343 (adjusted R squared = 0.311)					
Satisfaction (S)					
Task mental model quality (TMM)	18.651	1	18.651	8.610	0.004*
Task complexity (TC)	11.331	1	11.331	5.231	0.025*
Programming ability (GPA)	10.542	1	10.542	4.867	0.030*
TMM x TC	0.825	1	0.825	0.381	0.539
Error	177.631	82	2.166		
Total	1500.984	87			
Model R Squared = 0.147 (adjusted R squared = 0.105)					
Confidence in performance (CP)					
Task mental model quality (TMM)	12.347	1	12.347	17.112	0.000*
Task complexity (TC)	6.912	1	6.912	9.579	0.003*
Programming ability (GPA)	4.969	1	4.969	6.887	0.010*
TMM x TC	0.594	1	0.594	0.824	0.367
Error	59.167	82	0.722		
Total	77.924	87			
Model R squared = 0.235 (adjusted R squared = 0.197)					
***Significant at p=0.05*					

Table 2: Means and Standard Deviation for the Dependent Measures

Measures	Superior TMM			Inferior TMM			Total		
	Task complexity			Task complexity			Task complexity		
	Low	High	Mean	Low	High	Mean	Low	High	Mean
Software quality									
Mean	79.467	58.576	65.104	59.204	37.750	52.603	66.440	53.022	59.500
SD	31.309	29.593	31.369	24.635	23.956	26.115	28.568	29.447	29.635
n	15	33	48	27	12	39	42	45	87
Satisfaction									
Mean	4.575	3.966	4.156	3.755	2.875	3.484	4.048	3.675	3.855
SD	1.740	1.462	1.562	1.470	1.388	1.485	1.601	1.508	1.556
n	15	33	48	27	12	39	42	45	87
Confidence in performance									
Mean	0.482	0.024	0.167	-0.175	-0.888	-0.394	0.059	-0.219	-0.085
SD	1.028	0.851	0.924	0.931	0.574	0.894	1.006	0.881	0.948
n	15	33	48	27	12	39	42	45	87

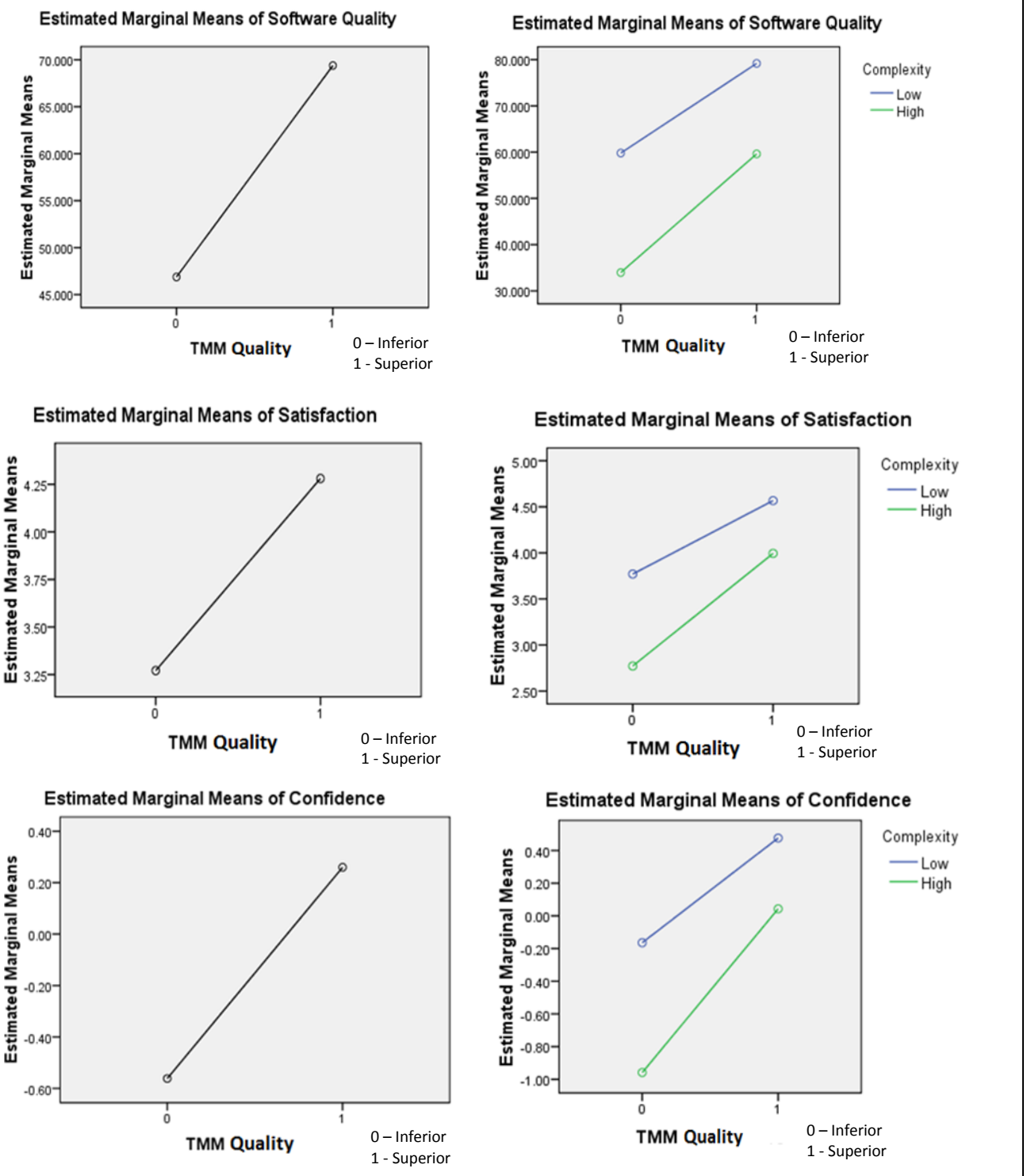

Figure 1: Marginal Means of Software Quality, Satisfaction, and Confidence in Performance

Software Quality

The ANCOVA analysis results (Table 1) indicate a significant main effect for TMM quality on software quality ($F(1,82) = 15.330$, $p < 0.01$). Hypothesis 1 predicted software quality to be higher for software developers or programming pairs with superior TMMs compared to those with inferior TMMs. Hypothesis 1 was supported because the marginal software quality scores of superior TMM group ($M = 69.30$) were significantly higher than those of inferior TMM group ($M = 45.291$) in a one-tailed test of marginal means ($p < 0.01$).

Hypothesis 4 predicted that there would be a significant interaction effect of task complexity on the relationship between TMM quality and software quality. Hypothesis 4 was not supported because the interaction of task complexity and TMM quality was not significant in the ANCOVA analysis ($F(1,82) = 0.295$, $p = 0.589$). Similarly, Hypothesis 5 predicted that there would be a significant interaction effect of programming setting on the relationship between TMM quality and software quality. As we discuss earlier, the preliminary 2 x 2 x 2 MANCOVA analysis suggested that interaction of TMM quality and programming setting was not significant. Thus, there is some evidence that Hypothesis 5 was not supported. However, as the number of observations in certain groups was too small for a rigorous 2 x 2 x 2 MANCOVA analysis, we grouped data for the two programming settings and carried out only a 2 x 2 MANCOVA analysis. Thus, we deem the evidence for rejecting Hypothesis 5 to be inconclusive because it could not be tested rigorously.

Satisfaction

The ANCOVA analysis results (Table 1) indicate a significant main effect for TMM quality on satisfaction ($F(1,82) = 8.610$, $p < 0.01$). Hypothesis 2 predicted that satisfaction of individual developers or collaborating pairs in the superior TMM group would be higher relative to the ones in the inferior TMM group. Hypothesis 2 was supported with marginal mean satisfaction reported by developers/pairs in the superior TMM group ($M = 4.342$) being significantly higher than that of the inferior TMM group ($M = 3.243$) in a one-tailed t-test of marginal means ($p < 0.01$). In addition, we found the interaction of TMM quality x task complexity on satisfaction to be not significant ($F(1,82) = 0.381$, $p = 0.539$)—we had no prior expectation on this interaction effect and, hence, did not include in our a priori hypotheses. Figure 1 graphically presents the marginal means for these conditions.

Confidence in Performance

The ANCOVA analysis results (Table 1) indicate a significant main effect of TMM quality on confidence in performance ($F(1,82) = 17.112$, $p < 0.01$). Hypothesis 3 predicted that confidence in performance of individual developers or pairs in the superior TMM group would be higher compared with those in the inferior TMM group. Hypothesis 3 was supported with marginal mean confidence in performance reported by the superior TMM group ($M = 0.288$) being significantly higher than that reported by inferior TMM group of developers ($M = -0.533$) in a one-tailed t-test of marginal means ($p < 0.01$). Also, we found the interaction of TMM quality x task complexity on confidence in performance to be not significant ($F(1,82) = 0.824$, $p = 0.367$))—again, we had no prior expectation on this interaction effect and, hence, did not include in our a priori hypotheses. Table 3 shows the marginal means and the results of comparison tests for Hypotheses 1, 2, and 3. Figure 1 graphically represents the marginal means for these conditions. Table 4 summarizes the results of hypothesis testing.

Table 3: Marginal Means and Planned Comparison Tests for Hypotheses 1, 2 and 3

Measure		Superior TMM	Inferior TMM	Hypotheses	p value
		1	2		
Software quality	SQ	69.39	46.88	H1: SQ1 – SQ2 > 0	< 0.01*
Satisfaction	S	4.28	3.27	H2: S1 – S2 > 0	< 0.01*
Confidence in performance	CP	0.26	-0.56	H3: CP1 – CP2 > 0	< 0.01*
**** Significant at p = 0.05***					

Table 4: Results of Hypotheses Testing

Hypothesis	Result
Software quality	
Hypothesis 1: *When working on a software maintenance task, an individual software developer or a collaborating pair of software developers with a superior TMM will achieve higher performance—measured in terms of software quality—compared to the an individual/pair without a superior TMM.*	Supported ($p < 0.01$)
Satisfaction	
Hypothesis 2: *When working on a software maintenance task, an individual software developer or a collaborating pair of software developers with a superior TMM will experience higher satisfaction compared to an individual/pair without a superior TMM.*	Supported ($p < 0.01$)
Confidence in performance	
Hypothesis 3: *When working on a software maintenance task, an individual software developer or a collaborating pair of software developers with a superior TMM has higher confidence in performance compared to an individual/pair*	Supported ($p < 0.01$)

without a superior TMM.	
Moderating effect of task complexity	
Hypothesis 4: *Task complexity will accentuate the software quality benefits for a software developer or a programming pair of software developers with a superior TMM.*	Not supported (p = 0.589)
Moderating effect of programming setting	
Hypothesis 5: *Paired software development setting, compared to individual software development setting, will accentuate software quality benefits for software developers with superior TMMs.*	Inconclusive

VI. DISCUSSION

Software development is a cognitively demanding task. Thus, a deeper understanding of the cognitive factors underlying software development would be helpful in enhancing software developers' performance. However, few research studies have investigated this phenomenon. Our research fills this void by emphasizing the role of mental models on software developers' performance under varying task conditions. Using a controlled laboratory experiment, we found that the quality of a developer's mental model positively impacted the individual's performance as measured in terms of software quality. Further, we found that this relationship between mental model quality and software quality persisted under varying task complexities.

Our main finding that superior mental models produced superior software quality is consistent with the notion of "programming as theory building" articulated by Naur (1985). Arguing for according primacy to knowledge acquisition by the programmers over mere production of program artifacts during systems development, Naur (1985) famously has argued that programming involves developers successfully building theories of how the system in question would handle the "affairs of the world" or help solve problems at hand. Such theories enable developers to not only comprehend how related laws apply to various aspects of their reality, but also help them recognize similar situations where these principles would apply. Further, developers who have an understanding of the theory of the system could easily articulate underlying rationale for the way the program is built. Thus, they have the ability to respond readily and constructively to any program modification requirements (Naur, 1985).

We argue that a developer's TMM captured at the end of a task reflects the "theories" the individual acquired about how the software handles the "affairs of the world" for which it is built. That is, having a superior TMM reflects a developer's building a more accurate theory of the reality embedded in the system. This finding is consistent with Naur's (1985) assertion that efficacious software is a byproduct of the manner in which developers marshal their knowledge and build theories of the reality embedded in a system.

A critical issue in furthering research on understanding mental models and their influence is the ability to elicit a mental model and assess its quality. Our conceptualization of a superior mental model that involves statistically comparing a mental model with a referent model provides a useful technique to evaluate mental model quality. This is an important methodological contribution of our research with implications for several research themes in the software development domain.

Implications for Research

Given our finding about performance benefits realized by software developers with superior TMMs, future research could further explore ways to help developers achieve superior TMMs during task performance. In turn, these results could help us refine software development methods and practices for achieving superior outcomes.

Because mental models are dynamic cognitive structures that capture a developer's state of comprehension at a given instant, the stage of the task performance at which they are measured matters. For instance, in this study, we measured TMM only once when the individual/pair completed their task to capture the individuals' final state of comprehension. This was consistent with our research questions and also quite appropriate for the short duration software maintenance tasks we used in our study. However, where appropriate, the measurement approach presented here could be used to measure mental models at multiple stages of task performance. For instance, evaluating a developer's mental model at two different points in time while they are performing a task is likely to provide insights into the incremental learning and the rate of evolution of the developer's comprehension.

Vandenbosch & Higgins (1996) suggest that insights into internal cognitions can further our understanding of how we learn. In a similar vein, Rowe & Cooke (1995) argue that a good understanding of mental models and their role in problem solving can help us with our pedagogy. Specifically, they suggest that mental models afford a glimpse into

the challenges that trainees encounter while learning something, and this awareness might enable us to anticipate training interventions that might facilitate better learning. For example, we could elicit the mental models of students in our classes, compare them with those of experts (i.e., professors), and make suitable changes to the content and delivery of our courses. This could be particularly useful in cognitively challenging courses such as programming and design that present a lot of learning difficulties to aspiring software developers.

Studies on the psychology of programming have endeavored to elucidate how programmers comprehend computer programs (Hoc, Green, Samurçay, & Gilmore, 1990; Pennington, 1987). These studies have revealed some of the strategies that programmers employ to understand programs, including top-down vs. bottom-up strategies (e.g., (Shneiderman & Mayer, 1979; Soloway & Ehrlich, 1984)) and control vs. functional flow strategies (e.g., Pennington, 1987). It would be interesting to see how mental models arising from these strategies (and, perhaps, moderated by task type) differ in quality.

While some researchers have argued that there is a relation between design patterns and schemata, and that design patterns influence the activation and/or generation of schemata (Kohls & Scheiter, 2008), the impact of design patterns on mental models has not been empirically demonstrated. If claims about the benefits of design patterns are indeed true, the use of patterns should result in superior mental models, which, in turn, should yield better performance. Our approach to measuring and testing mental models presented here could be very useful in this regard.

Yet another area in which our approach could be useful is in assessing mental models that emerge as a result of interactions between internal and external cognitive processes. Specifically, the theory of distributed cognition asserts that cognition is not just limited to internal processes, but may be distributed socially (e.g., through collaboration), structurally (e.g., embodied in external cognitive artifacts), or temporally (e.g., how previous cognitive experiences impinge on future cognitive events) (Flor & Hutchins, 1991; Hansen & Lyytinen, 2009; Hollan, Hutchins, & Kirsch, 2000). External representations, such as the unfolding solution to a problem or other artifacts that contain relevant information, have a bearing on team cognition (e.g., Rosen, Salas, Fiore, Pavlas, & Lum, 2009) and collaborative processes that lead to more effective problem-solving (e.g., Shirouzu, Miyake, & Masukawa, 2002). They also impact individuals' and groups' internal processes in ways that facilitate deeper understanding of the problem being solved. Given that cognitive externalization and other external resources can influence cognitive processes, we could assess their impact on mental models using the technique outlined in this study.

With this study demonstrates an empirical link between superior problem-comprehension (i.e., superior mental models) and software quality, future research could address how various methodologies and software practices could foster superior mental models and thereby facilitate software performance. For instance, in studying the efficacy of pair programming (Arisholm et al., 2007), the mental model measurement approach demonstrated in this study could help researchers investigate the cognitive benefits of different pairing methods (e.g., novice and expert pairing, assigned versus self-selected pairing, randomly assigned versus matched pairing based on personality/cognitive dispositions, etc.) and task settings (e.g., short- versus long-duration tasks, testing versus debugging tasks, design tasks with and without patterns, etc.).

Also, when investigating the efficacy of different tools and practices in agile methodologies, the mental model measurement approach outlined here could be helpful in identifying the critical levels of use of such practices (e.g., levels of initial design, test driven development, user involvement, etc.) for realizing optimum cognitive benefit to developers. Identifying such thresholds for optimum use of various practices, beyond which the law of diminishing returns (i.e., cognitive benefits) starts to kick in, should help further software practice.

Contrary to our expectation, task complexity did not moderate the relationship between superior TMM and software quality. Instead, task complexity had a main effect on the software quality achieved. That is, developers with superior TMM in the high-complexity task group achieved lower software quality than the ones in the low-complexity task group. With increasing task complexity, the performance of individuals/pairs could theoretically improve up to the limits of their cognitive capacity, beyond which performance deterioration could set in. We speculate that the high-complexity task used in our experimental setting may have exceeded this limit for many subjects, and thus may have adversely affected both TMM superior and TMM inferior groups. This needs further investigation in future research studies.

The moderating effect of programming setting (individual vs. paired programming) on the relationship between superior TMM and software quality achieved could not be conclusively tested due to the inadequacy of sample size in certain groups for doing a rigorous 2 x 2 x 2 MANCOVA analysis. We did not anticipate and control for this at the time of experimental design because TMM superiority was a derived factor. Since our study did not have sufficient

evidence to draw any conclusions on whether programming setting moderates the relationship between superior TMM and software quality, future studies could explore this relationship further by using larger sample sizes.

Implications for Practice

Being dynamic cognitive structures, mental models develop continuously over the duration of any task performance as developers learn and comprehend a task's requirements. With increased familiarity with the problem domain, an individual's mental model evolves as new connections are formed between domain concepts. As a developer's prior knowledge base is crucially important to help the individual learn new concepts (Hsu, 2006), it is reasonable to expect an experienced developer to have a superior TMM relative to a novice developer. Thus, all else being equal, a developer's experience should matter for developing a superior TMM and for achieving higher-quality software. This is consistent with and reinforces the conventional notion that software teams benefit from having at least some experienced developers in their midst.

Cognitive research highlights the inherent differences in the mental models developed by novices and experts. While the mental models of novices tend to be spotty and are limited to the surface features of problems, mental models of experts tend to be richer, abstract, integrative, and more stable (Davies, 1994; Glaser, 1989). Experts' mental models do contain several fragments that drive them to engage in knowledge-seeking activities (Sebrechts, Marsh, & Furstenburg, 1990). Compared with novices, experts, when confronted with a given situation, use richer connections and improved structure of their mental models to retrieve related knowledge from memory more rapidly and in larger chunks (Glaser, 1989).

This makes one wonder if there are ways novice developers could be helped to improve their TMMs. Human-computer interaction (HCI) research provides evidence that novice system users' mental models improve when they are provided with explicit models of the system (Sebrechts et al., 1990). During software development, modeling system requirements and specifications is a critical activity for understanding and communicating system specifications. Even in agile development methods, which emphasize minimal documentation, some modeling is always done relating to critical aspects of the system (e.g., formal UML models such as class diagrams, activity diagrams, sequence diagrams, etc.). Thus, it may be reasonable to speculate that novice developers would benefit more by having access to additional systems models beyond what may be needed by more experienced developers. Mindful of the "analysis-paralysis" trap, project managers and team leads may consider generating optimal levels of additional system models to help novices catch up quickly and enhance their contributions to projects.

Even informal models could also come in handy in facilitating this process of perspective making and perspective taking (Boland & Tenkasi, 1995) between experienced developers and novices. For instance, mind maps (i.e., graphical representations used for organizing information, where the most important concept is represented at the center and connected to other related concepts in a radial hierarchy) (Buzan & Buzan, 1993; Mahmud & Veneziano, 2011), influence diagrams (graphical models used to represent and solve complex decision problems under conditions of uncertain information) (Bielza, Gómez, & Shenoy, 2011; Howard & Matheson, 1981; Howard & Matheson, 2005), or various system or code maps developers typically sketch impromptu during the course of their daily work (DeLine, Venolia, & Rowan, 2010) could all be helpful towards this end.

Prior research in the HCI and education domains suggest that metaphors and analogies positively help subjects to develop mental models and foster performance in complex learning situations (Borgman, 1999; Cameron, 2002; Gentner & Gentner, 1983; Mayer, 1976; Streitz, Alfons, & Antonius, 1988). As per cognitive learning theory, metaphors enable subjects to quickly construct an initial mental model of a new domain by facilitating the mapping of concepts and interrelationships from a known domain to the new one. Subjects then use this initial mental model to test inferences and progressively refine it to make it more consistent with the new problem domain (Carroll & Thomas, 1982). Subjects may even use multiple analogies in constructing a mental model that help them draw different inferences relating to a target system (Gentner & Gentner, 1983). The XP practice of identifying system metaphors for documenting and communicating system functionality (e.g., shopping cart metaphor for online purchases, desktop metaphor for GUIs, assembly line metaphor for customer service, etc.) is highly recommended in agile development methods for multiple reasons (e.g., to create common vision among stakeholders, to provide shared vocabulary, to generate new ideas (problems and opportunities) about the system, and to help shape the system architecture by identifying key objects and their interface requirements) (Wake, 2000). In addition, based on evidence from the HCI and education research domains, system metaphors should positively help developers quickly build superior mental models of a system. Thus, they should be more critically integrated and emphasized when analyzing system requirements in project teams, irrespective of the development method used.

Study Limitations

Our research findings must be considered in light of our study's limitations. We used student subjects to understand the behavior of software developers. Our subjects reported to have, on average, 1.92 years of software development experience. Thus, the performance of our subjects may be comparable to those of entry-level professionals. Another limitation relates to the short durations of the experimental tasks. In practice, software developers are likely to engage with problems for periods longer than the two hours of our experimental setting. Therefore, future studies could explore how developers' mental models that evolved over longer periods of engagement with a problem affect software quality. Future research could also replicate the mental model measurement approach used here to help gain more confidence in the approach and the findings to further our understanding of the role of mental models in the software-development process.

As a way of experimental control, we blocked our participants' access to the Internet. Instead, we installed JDK API help documentation on their computers, which they could access during the experiment. In a real-world setting, developers typically scan Internet blogs, wikis, and other online resources for possible solutions to technical problems they may encounter while performing a programming task. Because we applied this experimental control to all participants in our study, we believe it has no impact on our main findings.

The two software-maintenance tasks we used in our research design were intellective in nature (i.e., tasks with correct answers) (Steiner, 1972). In such tasks, mental models of experts tend to converge. Thus, a consensual expert mental model could serve as a referent model for judging the quality of developers' mental models. However, when studying ill-structured tasks associated with software development (e.g., software design tasks), which tend to be less intellective and more judgmental (i.e., where there is no single right answer and the solution quality has to be judged consensually), the mental models are likely to diverge among experts and, thus, one single expert mental model would not be appropriate to serve as the referent model for comparison. One alternative would be to explore alternative possible solutions to the task and come up with multiple referent models. Future research could examine and extend the measurement approach illustrated here to unstructured and semi-structured tasks and settings, where there are multiple acceptable outcomes.

VII. CONCLUSION

The cognitive aspects of software development are of enduring interest to both practitioners and academics engaged in improving software-development processes and outcomes. Because developers' cognitive structures mediate the effects of various development practices, problem-solving approaches, and tools on the outcomes of software development (e.g.,. software quality), exploring such structures could provide us with insights into how developers process and store information when working on a software development task. This study explored one such cognitive structure (i.e., software developers' TMM when working in individual and paired development settings). Results of a controlled laboratory experiment suggest that the quality of a developer's TMM is a determinant of software quality achieved across two tasks of differing levels of complexity. This research presents an approach to measuring and evaluating mental models that provides a foundation for further research in the cognitive IS domain in general and software development in particular.

ACKNOWLEDGMENTS

This paper was funded in part by a summer research grant provided by the College of Business, Prairie View A&M University, Texas. An earlier version of this paper was presented at the HICSS conference (Balijepally, Nerur, & Mahapatra, 2012). We also thank the AE and the anonymous reviewers for their valuable comments and feedback that helped to significantly improve the paper.

REFERENCES

Editor's Note: The following reference list contains hyperlinks to World Wide Web pages. Readers who have the ability to access the Web directly from their word processor or are reading the paper on the Web, can gain direct access to these linked references. Readers are warned, however, that:

1. These links existed as of the date of publication but are not guaranteed to be working thereafter.
2. The contents of Web pages may change over time. Where version information is provided in the References, different versions may not contain the information or the conclusions referenced.
3. The author(s) of the Web pages, not AIS, is (are) responsible for the accuracy of their content.
4. The author(s) of this article, not AIS, is (are) responsible for the accuracy of the URL and version information.

Aiello, J. R., & Douthitt, E. A. (2001). Social facilitation from triplett to electronic performance monitoring. *Group Dynamics, 5*(3), 163-180.

Arisholm, E., Gallis, H., Dyba, T., & Sjoberg, D. I. K. (2007). Evaluating pair programming with respect to system complexity and programmer expertise. *IEEE Transactions on Software Engineering, 33*(2), 65-86.

Balijepally, V., Mahapatra, R., Nerur, S., & Price, K. H. (2009). Are two heads better than one for software development? The productivity paradox of pair programming. *MIS Quarterly, 33*(1), 91-118.

Balijepally, V., Nerur, S., & Mahapatra, R. (2012). Effect of task mental models on software developer's performance: An experimental investigation. *Proceedings of the 45th Hawaii International Conference on System Science,* 5442-5451.

Beck, K., & Andres, C. (2005). *Extreme programming explained: Embrace change*. Reading, MA:Addison-Wesley.

Bhattacherjee, A. (2001). Understanding information systems continuance: An expectation-confirmation model. *MIS Quarterly, 25*(3), 351-370.

Bielza, C., Gómez, M., & Shenoy, P. P. (2011). A review of representation issues and modeling challenges with influence diagrams. *Omega, 39*(3), 227-241.

Boland, R. J., & Tenkasi, R. V. (1995). Perspective making and perspective taking in communities of knowing. *Organization Science, 6*(4), 350-372.

Borgman, C. L. (1999). The user's mental model of an information retrieval system: An experiment on a prototype online catalog. *International Journal of Human-Computer Studies, 51*(2), 435-452.

Brewer, M. B., & Kramer, R. M. (1986). Choice behavior in social dilemmas: Effects of social identity, group size, and decision framing. *Journal of Personality and Social Psychology, 50*(3), 543-549.

Brooks, R. (1983). Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies, 18*(6), 543-554.

Burkhardt, J.-M., Détienne, F., & Wiedenbeck, S. (2002). Object-oriented program comprehension: Effect of expertise, task and phase. *Empirical Software Engineering, 7*(2), 115-156.

Buzan, T., & Buzan, B. (1993). *The mind map book: How to use radiant thinking to maximise your brain's untapped potential*. New York, NY: Plume.

Cameron, L. (2002). Metaphors in the learning of science: A discourse focus. *British Educational Research Journal*, *28*(5), 673-688.

Campbell, D. J. (1988). Task complexity: A review and analysis. *Academy of Management Review, 13*(1), 40-52.

Cañas, J. J., Bajo, M. T., & Gonzalvo, P. (1994). Mental models and computer programming. *International Journal of Human-Computer Studies, 40*(5), 795-811.

Carroll, J. M., & Thomas, J. C. (1982). Metaphor and the cognitive representation of computing systems. *IEEE Transactions on Systems, Man and Cybernetics, 12*(2), 107-116.

Collins, A., & Gentner, D. (1987). How people construct mental models. In D. Holland & N. Quinn (Eds.), *Cultural models in language and thought* (pp. 243-265). Cambridge: Cambridge University Press.

Davern, M., Shaft, T., & Te'eni, D. (2012). Cognition matters: Enduring questions in cognitive IS research. *Journal of the Association for Information Systems, 13*(4), 273-314.

Davies, S. P. (1994). Knowledge restructuring and the acquisition of programming expertise. *International Journal of Human-Computer Studies, 40*(4), 703-726.

DeLine, R., Venolia, G., & Rowan, K. (2010). Software development with code maps. *Commununications of the ACM, 53*(8), 48-54.

Doyle, J. K., & Ford, D. N. (1999). Mental models concepts revisited: Some clarifications and a reply to Lane. *System Dynamics Review, 15*(4), 411-415.

Durso, F. T., & Coggins, K. A. (1990). Graphs in the social and psychological sciences: Empirical contributions of Pathfinder. In R. W. Schvaneveldt (Ed.), *Pathfinder associative networks: Studies in knowledge organization* (pp. 31-51). Norwood, NJ: Ablex Publishing Corporation.

Edwards, B. D., Bell, S. T., Day, E. A., & Arthur, J. W. (2006). Relationships among team ability composition, team mental models, and team performance. *Journal of Applied Psychology, 91*(3), 727-736.

Flor, N. V., & Hutchins, E. L. (1991). Analyzing distributed cognition in software teams: A case study of team programming during perfective software maintenance. In J. Koenemann-Belleveau, T. G. Moher, & S. P.

Robertson (Eds.), *Proceedings of the Fourth Annual Workshop on Empirical Studies of Programmers* (pp. 36-63). Norwood, NJ: Ablex Publishing.

Gentner, D., & Gentner, D. (1983). Flowing waters or teeming crowds: Mental models of electricity. In D. Gentner & A. Stevens (Eds.), *Mental models* (pp. 99-130). Hillsdale, NJ: Lawrence Erlbaum Associates.

Gigerenzer, G., Hoffrage, U., & Kleinbölting, H. (1991). Probabilistic mental models: A Brunswikian theory of confidence. *Psychological Review, 98*(4), 506-528.

Glaser, R. (1989). Expertise and learning: How do we think about instructional processes now that we have discovered knowledge structures? In D. Klahr & K. Kotovshy (Eds.), *Complex information processing: The impact of Herbert A. Simon* (pp. 269-282). Hillsdale, NJ: Lawrence Erlbaum Associates.

Griffin, D., & Tversky, A. (1992). The weighing of evidence and the determinants of confidence. *Cognitive Psychology, 24*(3), 411-435.

Hair, J. F., Jr., Black, W. C., Babin, B. J., Anderson, R. E., & Tatham, R. L. (2006). *Multivariate data analysis* (6th ed). Upper Saddle River, NJ: Pearson Prentice-Hall.

Hansen, S., & Lyytinen, K. (2009). Distributed cognition in the management of design requirements. *Fifteenth Americas Conference on Information Systems San Francisco,* 1-10.

Harkins, S. G., & Petty, R. E. (1982). Effects of task difficulty and task uniqueness on social loafing. *Journal of Personality and Social Psychology, 43*(6), 1241-1229.

Helquist, J. H., Deokar, A., Meservy, T., & Kruse, J. (2011). Dynamic collaboration: Participant-driven agile processes for complex tasks. *The Database for Advances in Information Systems, 42*(2), 95-115.

Hoc, J.-M., Green, T. R. G., Samurçay, R., & Gilmore, D. J. (Eds.). (1990). *Psychology of programming*. London: Academic Press.

Hollan, J., Hutchins, E. L., & Kirsh, D. (2000). Distributed cognition: Toward a new foundation for human-computer interaction research. *ACM Transactions on Human-Computer Interaction, 7*(2), 174-196.

Howard, R. A., & Matheson, J. E. (1981). Influence diagrams. In R. A. Howard & J. E. Matheson (Eds.), *Readings on the principles and applications of decision analysis* (pp. 719-762). Menlo Park, CA: Strategic Decisions Group.

Howard, R. A., & Matheson, J. E. (2005). Influence diagrams. *Decision Analysis, 2*(3), 127-143.

Hsu, Y.-C. (2006). The effects of metaphors on novice and expert learners' performance and mental-model development. *Interacting with Computers, 18*(4), 770-792.

Hubert, L. (1985). Combinatorial data analysis: Association and partial association. *Psychometrika, 50*(4), 449-467.

Johnson-Laird, P. N. (1980). Mental models in cognitive science. *Cognitive Science, 4*(1), 71-115.

Johnson-Laird, P. N. (1983). *Mental models*. Cambridge, UK: Cambridge University Press.

Johnson-Laird, P. N. (1995). Mental models, deductive reasoning, and the brain. In M. S. Gazzaniga (Ed.), *The cognitive neurosciences* (pp. 999-1008). Cambridge, MA: MIT Press.

Jourden, F. J., & Heath C. (1996). The evaluation gap in performance perceptions: Illusory perceptions of groups and individuals. *Journal of Applied Psychology, 81*(4), 369-379.

Karau, S. J., & Williams, K. D. (1993). Social loafing: A meta-analytic review and theoretical integration. *Journal of Personality and Social Psychology, 65*(4), 681-706.

Kohls, C., & Scheiter, K. (2008). The relation between design patterns and schema theory. *Proceedings of the 15th Conference on Pattern Languages of Programs, Nashville, Tennessee,* 1-16.

Koriat, A., Lichtenstein, S., & Fischhoff, B. (1980). Reasons for confidence. *Journal of Experimental Psychology: Human Learning and Memory, 6*(2), 107-118.

Krackhardt, D., & Porter, L. W. (1986). The snowball effect: Turnover embedded in communication networks. *Journal of Applied Psychology, 71*(1), 50-55.

Kudikyala, U. K., & Vaughn, R. B. (2005). Software requirement understanding using Pathfinder networks: Discovering and evaluating mental models. *Journal of Systems and Software, 74*(1), 101-108.

Langan-Fox, J., Anglim, J., & Wilson, J. R. (2004). Mental models, team mental models, and performance: Process, development, and future directions. *Human Factors and Ergonomics in Manufacturing, 14*(4), 331-352.

Laughlin, P. R. (1980). Social combination processes of cooperative problem-solving groups on verbal intellective tasks. In M. Fishbein (Ed.), *Progress in social psychology* (pp. 127-155). Hillsdale, NJ: Erlbaum.

Letovsky, S. (1987). Cognitive processes in program comprehension. *Journal of Systems and Software, 7*(4), 325-339.

Littman, D. C., Pinto, J., Letovsky, S., & Soloway, E. (1987). Mental models and software maintenance. *Journal of Systems and Software, 7*(4), 341-355.

Mahmud, I., & Veneziano, V. (2011). Mind-mapping: An effective technique to facilitate requirements engineering in agile software development. *14th International Conference on Computer and Information Technology,* 157-162.

Mathieu, J. E., Heffner, T. S., Goodwin, G. F., Salas, E., & Cannon-Bowers J. A. (2000). The influence of shared mental models on team process and performance. *Journal of Applied Psychology, 85*(2), 273-283.

Mayer, R. E. (1976). Some conditions of meaningful learning for computer programming: Advance organizers and subject control of frame order. *Journal of Educational Psychology, 68*(2), 143-150.

McKenzie, C. R. M. (1997). Underweighting alternatives and overconfidence. *Organizational Behavior and Human Decision Processes, 71*(2), 141-160.

Naur, P. (1985). Programming as theory building. *Microprocessing and Microprogramming, 15*(5), 253-261.

Nawrocki, J., & Wojciechowski, A. (2001). Experimental evaluation of pair programming. *12th European Software Control and Metrics Conference,* 269-276.

Neter, J., Kutner, M. H., Nachtsheim, C. J., & Wasserman, W. (1996). *Applied linear models* (4th ed.). Chicago: Irvin.

Newell, A., & Simon, H. A. (1972). *Human problem solving*. Englewood Cliffs, NJ: Prentice-Hall.

Nosek, J. T. (1998). The case for collaborative programming. *Communications of the ACM, 41*(3), 105-108.

Pennington, N. (1987). Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive Psychology, 19*(3), 295-341.

Petty, R., E., Harkins, S. G., Williams, K. D., & Latane, B. (1977). The effects of group size on cognitive effort and evaluation. *Personality and Social Psychology Bulletin, 3*(4), 575-578.

Prell, C. (2012). *Social network analysis: History, theory & methodology*. Thousand Oaks, CA: Sage.

Rentsch, J. R., & Hall, R. J. (1994). Members of great teams think alike: A model of team effectiveness and schema similarity among team members. In M. M. Beyerlein & D. A. Johnson (Eds.), *Advances in interdisciplinary studies of work teams: Theories of self-managing work* teams (Vol. 1., pp. 223-261). US: JAI Press.

Rentsch, J. R., & Klimoski, R. J. (2001). Why do "great minds" think alike?: Antecedents of team member schema agreement. *Journal of Organizational Behavior, 22*(2), 107-120.

Rentsch, J. R., & Woehr, D. J. (2004). Quantifying congruence in cognition: Social relations modeling and team member schema similarity. In E. Salas and S. M. Fiore (Eds.), *Team cognition: Understanding the factors that drive process and performance* (pp. 11-31). Washington, DC: American Psychological Association.

Reusser, K. (1990). From text to situation to equation: Cognitive simulation of understanding and solving mathematical word problems. In H. Mandl, E. DeCorte, N. Bennett, & H. F. Friedrich (Eds.), *Learning and instruction: Analysis of complex skills and complex knowledge domains* (pp. 477-498). Elmsford, NY: Pergamon Press.

Riley, M. S., & Greeno, J. G. (1988). Developmental analysis of understanding language about quantities and of solving problems. *Cognition & Instruction, 5*(1), 49-101.

Robillard, P. N. (1999). The role of knowledge in software development. *Communications of the ACM, 42*(1), 87-92.

Rosen, M. A., Salas, E., Fiore, S. M., Pavlas, D., & Lum, H. C. (2009). Team cognition and external representations: A framework and propositions for supporting collaborative problem solving. *Proceedings of the Human Factors and Ergonomics Society Annual Meeting, 53*(18), 1295-1299.

Rowe, A. L., & Cooke, N. J. (1995). Measuring mental models: Choosing the right tools for the job. *Human Resource Development Quarterly, 6*(3), 243-255.

Santhanam, R., & Sein, M. K. (1994). Improving end-user proficiency: Effects of conceptual training and nature of interaction. *Information Systems Research, 5*(4), 378-399.

Sebrechts, M. M., Marsh, R. L., & Furstenburg, C. T. (1990). Integrative modeling: Changes in mental models during learning. In P. R. Scott, W. Z. Wayne, & B. B. John (Eds.), *Cognition, computing, and cooperation* (pp. 338-398). Norwood, NJ: Ablex Publishing Corporation.

Shaft, T. M., & Vessey, I. (1995). The relevance of application domain knowledge: The case of computer program comprehension. *Information Systems Research, 6*(3), 286-299.

Shaft, T. M., & Vessey, I. (1998). The relevance of application domain knowledge: Characterizing the computer program comprehension process. *Journal of Management Information Systems, 15*(1), 51-78.

Shaft, T. M., & Vessey, I. (2006). The role of cognitive fit in the relationship between software comprehension and modification. *MIS Quarterly, 30*(1), 29-55.

Shih, Y.-F., & Alessi, S. M. (1993). Mental models and transfer of learning in computer programming. *Journal of Research on Computing in Education, 26*(2), 154-175.

Shirouzu, H., Miyake, N., & Masukawa, H. (2002). Cognitively active externalization for situated reflection. *Cognitive Science, 26*(4), 469-501.

Shneiderman, B., & Mayer, R. (1979). Syntactic/semantic interactions in programmer behavior: A model and experimental results. *International Journal of Parallel Programming, 8*(3), 219-238.

Simpson, W. (2001). *QAP: The quadratic assignment procedure*. Paper presented at the American Stata Users' Group Meeting. Retrieved from http://fmwww.bc.edu/RePEc/nasug2001/simpson.pdf

Soloway, E., & Ehrlich, K. (1984). Empirical studies of programming knowledge. *IEEE Transactions on Software Engineering,* SE-10(5), 595-609.

Steiner, I. D. (1972). *Group process and productivity*. New York: Academic Press.

Stern, E. (1993). What makes certain arithmetic word problems involving the comparison of sets so difficult for children? *Journal of Educational Psychology, 85*(1), 7-23.

Storey, M. A. D., Fracchia, F. D., & Muller, H. A. (1997). Cognitive design elements to support the construction of a mental model during software visualization. *Fifth International Workshop on Program Comprehension,* 17-28.

Streitz, N. A., Alfons, L., & Antonius, W. (1988). The combined effects of metaphor worlds and dialogue modes in human-computer interaction. In F. Klix, N. Streitz, Y. Waern, & H. Wandke (Eds.), *Man-computer interaction research* (pp. 75-88). Amsterdam: Elsevier.

Stubbart, C. I. (1989). Managerial cognition: A missing link in strategic management research. *Journal of Management Studies, 26*(4), 325-347.

Tarafdar, M., Tu, Q., & Ragu-Nathan, T. S. (2010). Impact of technostress on end-user satisfaction and performance. *Journal of Management Information Systems, 27*(3), 303-334.

Vandenbosch, B., & Higgins, C. (1996). Information acquisition and mental models: An investigation into the relationship between behaviour and learning. *Information Systems Research, 7*(2), 198-214.

Wake, B. (2000). *The system metaphor.* Retrieved from http://xp123.com/articles/the-system-metaphor/

Williams, K. D., Harkins, S. G., & Latane, B. (1981). Identifiability as a deterrent to social loafing: Two cheering experiments. *Journal of Personality and Social Psychology, 40*(2), 303-311.

Williams, L. (2000). *The collaborative software process* (PhD dissertation). Department of Computer Science, University of Utah.

Williams, L., A., & Kessler, R. R. (2000). All I really need to know about pair programming I learned in kindergarten. *Communications of the ACM, 43*(5), 108-114.

Wilson, J. R. (2000). Mental models. In W. Karwowski (Ed.), *International encyclopedia of ergonomics and human factors* (pp. 493-496). London: Taylor and Francis.

Wilson, J. R., & Rutherford, A. (1989). Mental models: Theory and application in human factors. *Human Factors, 31*(6), 617-634.

Zajonc, R. B. (1965). Social facilitation. *Science,* 149(3681), 269-274.

Zhang, C., & Budgen, D. (2012). What do we know about the effectiveness of software design patterns. *IEEE Transactions on Software Engineering, 38*(5), 1213-1231.

APPENDIX A: TASK MENTAL MODEL QUESTIONS FOR LOW-COMPLEXITY TASK

Please answer the following questions based on your understanding of the programming task that you had just completed. Please indicate your perception of how closely related are the following classes and methods of the programming task. Use a rating scale from 1—not at all related to 7—highly related.

Student class	StudentTest class	1 2 3 4 5 6 7
Student class	getScores ( )	1 2 3 4 5 6 7
Student class	computeAverage ( )	1 2 3 4 5 6 7
Student class	computeGrade ( )	1 2 3 4 5 6 7
Student class	toString ( )	1 2 3 4 5 6 7
Student class	main ( )	1 2 3 4 5 6 7
StudentTest class	getScores ( )	1 2 3 4 5 6 7
StudentTest class	computeAverage ( )	1 2 3 4 5 6 7
StudentTest class	computeGrade ( )	1 2 3 4 5 6 7
StudentTest class	toString ( )	1 2 3 4 5 6 7
StudentTest class	main ( )	1 2 3 4 5 6 7
getScores ( )	computeAverage ( )	1 2 3 4 5 6 7
getScores ( )	computeGrade ( )	1 2 3 4 5 6 7
getScores ( )	toString ( )	1 2 3 4 5 6 7
getScores ( )	main ( )	1 2 3 4 5 6 7
computeAverage ( )	computeGrade ( )	1 2 3 4 5 6 7
computeAverage ( )	toString ( )	1 2 3 4 5 6 7
computeAverage ( )	main ( )	1 2 3 4 5 6 7
computeGrade ( )	toString ( )	1 2 3 4 5 6 7
computeGrade ( )	main ( )	1 2 3 4 5 6 7
toString ( )	main ( )	1 2 3 4 5 6 7

The above questions capture the perceived relationships between the following seven classes/methods of the student grades application:

1. Student class
2. StudentTest class
3. getScores()
4. computeAverage()
5. computeGrade()
6. toString()
7. main()

Two experts jointly identified these classes/methods as the most important concepts associated with the experimental task. The perceived strengths of relationships among these concepts reflect a developer's understanding of the programming task

APPENDIX B: TASK MENTAL MODEL QUESTIONS FOR HIGH-COMPLEXITY TASK

Please answer the following questions based on your understanding of the programming task that you had just completed. Please indicate your perception of how closely related are the following classes and methods of the programming task. Use a rating scale from 1—not at all related to 7—highly related.

Application	Movie	1 2 3 4 5 6 7
Application	MovieCollection	1 2 3 4 5 6 7
Application	UserInterface	1 2 3 4 5 6 7
Application	getTitle ( )	1 2 3 4 5 6 7
Application	add ( )	1 2 3 4 5 6 7
Application	run ( )	1 2 3 4 5 6 7
Application	addMovie ( )	1 2 3 4 5 6 7
Application	displayMovie ( )	1 2 3 4 5 6 7
Application	menu ( )*	1 2 3 4 5 6 7
Movie	MovieCollection	1 2 3 4 5 6 7
Movie	UserInterface	1 2 3 4 5 6 7
Movie	getTitle ( )	1 2 3 4 5 6 7
Movie	add ( )	1 2 3 4 5 6 7
Movie	run ( )	1 2 3 4 5 6 7

Movie	addMovie ( )	1	2	3	4	5	6	7
Movie	displayMovie ( )	1	2	3	4	5	6	7
Movie	menu ( )*	1	2	3	4	5	6	7
MovieCollection	UserInterface	1	2	3	4	5	6	7
MovieCollection	getTitle ( )	1	2	3	4	5	6	7
MovieCollection	add ( )	1	2	3	4	5	6	7
MovieCollection	run ( )	1	2	3	4	5	6	7
MovieCollection	addMovie ( )	1	2	3	4	5	6	7
MovieCollection	displayMovie ( )	1	2	3	4	5	6	7
MovieCollection	menu ( )*	1	2	3	4	5	6	7
UserInterface	getTitle ( )	1	2	3	4	5	6	7
UserInterface	add ( )	1	2	3	4	5	6	7
UserInterface	run ( )	1	2	3	4	5	6	7
UserInterface	addMovie ( )	1	2	3	4	5	6	7
UserInterface	displayMovie ( )	1	2	3	4	5	6	7
UserInterface	menu ( )*	1	2	3	4	5	6	7
getTitle ( )	add ( )	1	2	3	4	5	6	7
getTitle ( )	run ( )	1	2	3	4	5	6	7
getTitle ( )	addMovie ( )	1	2	3	4	5	6	7
getTitle ( )	displayMovie ( )	1	2	3	4	5	6	7
getTitle ( )	menu ( )*	1	2	3	4	5	6	7
add ( )	run ( )	1	2	3	4	5	6	7
add ( )	addMovie ( )	1	2	3	4	5	6	7
add ( )	displayMovie ( )	1	2	3	4	5	6	7
add ( )	menu ( )*	1	2	3	4	5	6	7
run ( )	addMovie ( )	1	2	3	4	5	6	7
run ( )	displayMovie ( )	1	2	3	4	5	6	7
run ( )	menu ( )*	1	2	3	4	5	6	7
addMovie ( )	displayMovie ( )	1	2	3	4	5	6	7
addMovie ( )	menu ( )*	1	2	3	4	5	6	7
displayMovie ( )	menu ( )*	1	2	3	4	5	6	7

The above questions capture the perceived relationships among the following ten classes/methods of the movie rental application:

1. Application class
2. Movie
3. MovieCollection
4. UserInterface
5. getTitle ( )
6. add ( )
7. run ( )
8. addMovie ( )
9. displayMovie ( )
10. menu ( )*

Two experts jointly identified these classes/methods as the most important concepts associated with the experimental task. The perceived strengths of relationships among these concept reflects a developer's understanding of the programming task.

ABOUT THE AUTHORS

VenuGopal Balijepally is an Assistant Professor of MIS in the School of Business Administration at Oakland University, Michigan. He received his PhD in information systems from the University of Texas at Arlington and post-graduate diploma in management (MBA), from the Management Development Institute, Gurgaon, India. He also holds a master's degree from Indian Institute of Technology, Mumbai and a bachelor's degree from Osmania University, India, both in civil engineering. His research interests include software development, social capital of IS teams, knowledge management, IT management, and research methods. His research publications appear in *MIS Quarterly*, *Journal of International Business* Studies, *Journal of the AIS*, *Communications of the ACM*, *Communications of the AIS*, *Journal of Systems and Software*, *Scientometrics*, and various conference proceedings such as the Americas Conference on Information Systems, the Hawaii International Conference on System Sciences, and the Decision Sciences Institute.

Sridhar Nerur is a Professor of Information Systems at the University of Texas at Arlington. He holds an engineering degree in electronics from Bangalore University, a PGDM (MBA) from the Indian Institute of Management, Bangalore, India, and a PhD in business administration from the University of Texas at Arlington. His research appears in *MIS Quarterly, Strategic Management Journal, Communications of the ACM, Communications of the AIS, The DATA BASE for Advances in Information Systems, European Journal of Information Systems, Information Systems Management,* and *Journal of International Business Studies*. He has served as an associate editor for the *European Journal of Information Systems*. His research and teaching interests consider social networks, software design, adoption of software development methodologies, cognitive aspects of programming, dynamic IT capabilities, and agile software development.

RadhaKanta Mahapatra is a Professor of Information Systems at the University of Texas at Arlington. He holds a bachelor's degree in Electrical Engineering from the National Institute of Technology, Rourkela, India, a PGDM (MBA) from the Indian Institute of Management, Ahmedabad, India, and a PhD in Information Systems from Texas A&M University. His research interests include agile software development and project management, data warehousing and business intelligence, data quality, and healthcare information systems. His research publications have appeared in *MIS Quarterly, Communications of the ACM, Decision Support Systems, Information & Management, European Journal of Information Systems*, and other journals. He received the Distinguished Research Publication Award and the Distinguished Professional Publication Award from the College of Business Administration of the University of Texas at Arlington.

Communications of the Association for Information Systems

ISSN: 1529-3181

Communications of the Association for Information Systems