EFFECTS OF DIAGRAMMATIC REPRESENTATION ON SOFTWARE EVOLUTION
PROGRAMMING PERFORMANCE—AN EXPERIMENTAL INVESTIGATION
OF UML DIAGRAMS

by

LULU ZHANG

Presented to the Faculty of the Graduate School of
The University of Texas at Arlington in Partial Fulfillment
of the Requirements
for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT ARLINGTON
August 2016
Acknowledgments

I would like to express my sincere gratitude to my advisors, Dr. Radha Mahapatra and Dr. Sridhar Nerur, for their patient guidance and mentorship during my PhD study. They have taught me how to approach the work as a researcher. Their extensive knowledge and systematic way of thinking have been exceptionally valuable to me. Their specific and constructive comments throughout the dissertation process provided the basis for this study. I am truly fortunate to have had the opportunity to learn from them. More important, their understanding, encouragement, and care helped me get through some tough moments when I got lost in some negative thoughts. They always believe in me and never gave up on me. I am grateful to them. I would like to thank Dr. Mary Whiteside for her invaluable advice and thought-provoking suggestions for my dissertation. I admire not only her intelligence but also her elegance and grace.

Dr. James T.C. Teng was an encouraging mentor even before I started my doctoral journey. He encouraged me to pursue my PhD when I was still a master's student. I would not have had such a great opportunity to work with so many intelligent people without him. He is knowledgeable and open-minded and gave me great inspiration and excellent guidance during both my course- and research work.

I am grateful to my colleagues, YoonSang, Kriti, Indika, Philip, Rashid, Ola, and Ganapathiraman, for their help with the experiments in this dissertation. Special thanks goes to David Leong for his assistance in setting up the experiment in the lab. I also want to thank my office mates and dear friends, Dr. Carol J. Cagle and Deanna M. House. Our academic discussions and personal conversations are positive memories I will never forget. I learned a lot from them, and I am full of appreciation for their help. I feel very lucky to have had them as companions on my PhD journey.
Last but not least, I would like to dedicate my dissertation to my parents and thank them for their unconditional support and love. This dissertation is also dedicated to my dear husband, Qi. I thank him with all my heart for his constant love, support, and considerate care of me.

August 11, 2016
Abstract

EFFECTS OF DIAGRAMMATIC REPRESENTATION ON SOFTWARE EVOLUTION PROGRAMMING PERFORMANCE—AN EXPERIMENTAL INVESTIGATION OF UML DIAGRAMS

Lulu Zhang, PhD

The University of Texas at Arlington, 2016

Supervising Professors: Radha Mahapatra and Sridhar Nerur.

The fundamental question of the present study is to investigate whether fit between representation and task type matters while developing software. The UML (Unified Modeling Language) includes diagrams that allow developers to represent software artifacts at different levels of abstraction. These diagrams facilitate design, specification, and testing during the software development life circle. However, they are expensive and difficult artifacts to maintain because of the time pressure that developers in the software industry typically experience. The present study hypothesizes that the selective usage of diagrammatic representations would justify the cost of maintaining these representations. Student subjects participated in controlled laboratory experiments in which two factors were manipulated: task types and diagram types. The subjects’ comprehension of the system being developed as well as their coding performance were compared across experimental treatments. The results indicate that the effectiveness of diagram usage depends on the task type. The fit between diagram and task type has a positive relationship with both comprehension and coding quality performance. However, comprehension does not always fully mediate the relationship between fit and coding
performance. The findings suggest that developers do not need all types of UML diagrams, but would benefit by choosing the ones that are compatible with the task at hand. Implications of the study for practitioners and academics are discussed as well.
# Table of Contents

Acknowledgments ........................................................................................................ iii

Abstract ..................................................................................................................... v

List of Illustrations .................................................................................................. xii

List of Tables .......................................................................................................... xiii

Chapter 1 Introduction .......................................................................................... 1
  1.1 Importance of Research ................................................................................... 5
  1.2 Comparison to Existing Similar Studies ......................................................... 8
  1.3 Research Questions .......................................................................................... 12
  1.4 Overview .......................................................................................................... 12

Chapter 2 Literature Review ................................................................................. 13
  2.1 Software Maintenance .................................................................................... 13
    2.1.1 Review of Software Maintenance ............................................................ 13
    2.1.2 Perfective Maintenance .......................................................................... 21
    2.1.3 Complexity Issues with OO Programming Language ......................... 22
    2.1.4 Documentation Issues with OO and Agile Methodologies .................. 24
    2.1.5 Summary ................................................................................................. 26
  2.2 Conceptual Modeling ...................................................................................... 26
    2.2.1 Conceptual Modeling and Comprehension .............................................. 27
      2.2.1.1 Modeling Notations .......................................................................... 27
      2.2.1.2 Modeling Approaches and Tools ...................................................... 28
      2.2.1.3 Modeling Context ........................................................................... 30
      2.2.1.4 Summary of implications ................................................................. 32
    2.2.2 Conceptual Modeling and Performance .................................................. 32
    2.2.3 Conceptual Modeling Representation ...................................................... 33
4.5 Response Variable Measurements .............................................................................. 80
  4.5.1 Comprehension Measures .................................................................................. 80
  4.5.2 Coding Performance Measure .......................................................................... 81
4.6 Statistical Analysis .................................................................................................... 82
Chapter 5 Research Results .............................................................................................. 86
  5.1 Preliminary Analyses ............................................................................................... 86
    5.1.1 Sample Characteristics .................................................................................... 86
    5.1.2 Assumptions ..................................................................................................... 88
    5.1.3 Reliability of Dependent Measures .................................................................. 90
    5.1.4 Test of Significance—Main Effects .................................................................. 91
    5.1.5 Test of Significance—Correlations .................................................................. 94
  5.2 Hypotheses Testing ................................................................................................. 97
    5.2.1 Hypotheses Concerning Comprehension ......................................................... 98
      5.2.1.1 Task1: Structural Programming Task (H1a, 1b, 1c) ................................. 98
        5.2.1.1.1 Hypothesis H1a .................................................................................... 99
        5.2.1.1.2 Hypothesis H1b .................................................................................. 100
        5.2.1.1.3 Hypothesis H1c .................................................................................. 101
      5.2.1.2 Task 2: Behavioral Programming task (H2a, 2b, 2c) .............................. 102
        5.2.1.2.1 Hypothesis H2a .................................................................................. 102
        5.2.1.2.2 Hypothesis H2b .................................................................................. 103
        5.2.1.2.3 Hypothesis H2c .................................................................................. 104
    5.2.2 Hypotheses concerning Code Quality ............................................................... 105
      5.2.2.1 Task1: Structural Programming Task (H3a, 3b, 3c) ................................. 105
        5.2.2.1.1 Hypothesis H3a .................................................................................. 106
        5.2.2.1.2 Hypothesis H3b .................................................................................. 107
5.2.2.1.3 Hypothesis H3c

5.2.2.2 Task 2: Behavioral Programming Task (H4a, 4b, 4c)

5.2.2.2.1 Hypothesis H4a

5.2.2.2.2 Hypothesis H4b

5.2.2.2.3 Hypothesis H4c

5.2.3 Hypotheses concerning Mediation Effect of Comprehension (H5a, 5b)

5.2.3.1 Hypothesis 5a

5.2.3.2 Hypothesis 5b

5.2.4 Summary of Hypotheses Test

Chapter 6 Summary, Limitations, and Future Research

6.1 Summary of Research Findings

6.1.1 Regarding the Comprehension

6.1.2 Regarding the Coding Quality

6.1.3 Regarding the Mediation Effect of Comprehension

6.1.4 Summary of Findings

6.2 Significance of Findings

6.3 Implications to practitioners

6.4 Limitations

Appendix A Enrollment Flyer and Informed Consent

Appendix B Debriefing

Appendix C Warm-up Task Descriptions and Requirements

Appendix D Warm-up Task—UML Diagrams

Appendix E Task 1: Structural Task—Description

Appendix F Task 1: Structural Task—Modification Requirements
List of Illustrations

Figure 2-1 Example of Class diagram: Banking System ........................................... 42
Figure 2-2 Example of Sequence Diagram: Deposit Transaction ................................. 44
Figure 2-3 Example of Sequence Diagram: Withdrawal Transaction ............................. 45
Figure 2-4 Example of Communication Diagram: Deposit Transaction .......................... 47
Figure 2-5 Process to Increase Program Performance ............................................... 52
Figure 2-6 Process to Increase Program Performance—Distributed Task ...................... 54
Figure 2-7 Process to Increase Program Performance—FIT ....................................... 57
Figure 2-8 Process to Increase Program Performance—Direct Effect of FIT .................. 58
Figure 2-9 Research Model—Structural Task ........................................................... 59
Figure 5-1 Marginal Means of Structural Comprehension Score—Structural Task ........... 100
Figure 5-2 Marginal Means of Structural Comprehension Score—Behavioral Task .......... 103
Figure 5-3 Marginal Means of Coding Quality—Structural Task ................................ 107
Figure 5-4 Marginal Means of Coding Quality—Behavioral Task ................................ 109
Figure 5-5 Mediator Test Models .............................................................................. 112
List of Tables

Table 1-1 Categories of Similar Studies ................................................................. 10
Table 2-1 Summary of Studies Regarding Software Maintenance Issues .................... 16
Table 2-2 Notations of Class Diagram ........................................................................ 40
Table 2-3 Notations of Sequence Diagram ................................................................... 43
Table 2-4 Notations of Communication Diagram ......................................................... 46
Table 3-1 Cognitive Fit ............................................................................................... 63
Table 3-2 Summary of Hypotheses ............................................................................ 75
Table 4-1 Experiment Design ..................................................................................... 78
Table 5-1 Distributions of Subjects Across Treatments .............................................. 86
Table 5-2 One-way ANOVA Results on Semesters ................................................... 87
Table 5-3 Demographics Count Frequency by Treatment ($N = 99$) ....................... 88
Table 5-4 Assumptions of Normality—Shapiro-Wilk $p$-value .................................. 89
Table 5-5 Assumption of Equal Variance—Leven $p$-value ....................................... 90
Table 5-6 MANOVA Label: Type of FIT .................................................................. 92
Table 5-7 MANCOVA Results for Dependent Variables ......................................... 93
Table 5-8 One-Way ANOVA Results on Dependent Variables—Factor Level .......... 94
Table 5-9 Dummy Variables ...................................................................................... 95
Table 5-10 Significant of Correlation ....................................................................... 96
Table 5-11 Tukey-Kramer Comparison—Structural Comprehension Scores ($\alpha = 0.05$) 99
Table 5-12 Means for Structural Comprehension Score—$H1a$ ................................. 99
Table 5-13 Means for Structural Comprehension Score—$H1b$ ................................. 101
Table 5-14 Means for Structural Comprehension Score—$H1c$ ................................. 101
Table 5-15 Tukey-Kramer Comparison of Structural Comprehension Scores ($\alpha = 0.05$) 102
Table 5-16 Means for Structural Comprehension Score—H2a ................................................. 102
Table 5-17 Means for Structural Comprehension Score—H2b ................................................. 104
Table 5-18 Means for Structural Comprehension Score—H2c ................................................. 104
Table 5-19 Tukey-Kramer Comparison Report of Average Coding Quality (α = 0.05) .................. 106
Table 5-20 Means for Coding Quality—H3a .................................................................................. 106
Table 5-21 Means for Coding Quality—H3b .................................................................................. 107
Table 5-22 Means for Coding Quality—H3c .................................................................................. 108
Table 5-23 Tukey-Kramer Comparison Report of Coding Quality (α= 0.05) ......................... 109
Table 5-24 Means for Coding Quality—H4a .................................................................................. 109
Table 5-25 Means for Coding Quality—H4b .................................................................................. 110
Table 5-26 Means for Coding Quality—H4c .................................................................................. 110
Table 5-27 Summary of Regression Analysis—H5a ................................................................. 114
Table 5-28 Summary of Regression Analysis—H3b ................................................................. 115
Table 5-29 Results of Hypotheses Testing ...................................................................................... 116
Chapter 1

Introduction

Software maintenance is a critical phase in the software development life cycle. Total system maintenance costs are estimated to be about 70 percent of all costs incurred on software systems (Lientz et al. 1980).

Software maintenance is not limited to error corrections. It also refers to any modification that must be made to the software after it has been delivered. There are four types of maintenance tasks: corrective, adaptive, perfective, and preventive (Standardization 2006). Among the four types, only corrective maintenance is in the realm of traditional maintenance for correcting errors and optimizing performance. The other three types are in the domain of software evolution. The term evolution has been used to characterize the growth dynamics of software from the early 1960s (Chapin et al. 2001). Today, approximately 75 percent of maintenance costs are spent on evolving systems in the form of adaptive and perfective maintenance (Van Vliet 2000). Therefore, to narrow the topic and focus effort on the most salient issues, this study investigates how to improve software evolution performance with minimal time and effort.

The need to improve software evolution performance with minimal effort and cost has prompted the software engineering community to implement various initiatives. One initiative is to reduce knowledge transfer efforts (Lim et al. 2005). The developers responsible for software evolution tasks often did not participate in the original software design (Arisholm et al. 2006). Once the software is handed over to other developers for maintenance, the new developers first consult with the software documentation. If the documentation is not available or is not well written, the developers will struggle to perform evolution tasks efficiently. Therefore, documentation tools have been advocated
to help reduce maintainers’ comprehension efforts and time while changing complex systems (Booch et al. 1999; Larman 2004).

In practice, however, software documentation is inadequate and incomplete in most software projects (Briand 2003). The main reason is that developers do not perceive the benefits of documentation. The developers who produced the documentation are typically not its final users. The final users could be the other team members during the analysis and design phase or the maintainer during the maintenance phase. Thus, developers often perceive documentation as a time-consuming and useless effort, particularly when time is of the essence. Developers can achieve sound short-term business goals with inadequate documentation, but will ultimately leave potential hurdles to be handled through collaboration among team members or system enhancement during the maintenance phase.

Lack of documentation has become severe with the emergence of new software development approaches such as the Agile methodology. One of the core principles of Agile Software Development (ASD) is keeping documentation to a minimum and focusing on test cases as a source of system requirements (Ambler 2002). Proponents of ASD advocate performing maintenance tasks in rapid iterations. The Agile method promotes collaborations, which elicit proper software maintenance needs, produce frequent software releases, leverage communications, and respond to rapidly changing needs (Svensson et al. 2005). Developers perform maintenance tasks in a short period with rapid and close interactions with clients. Developers keep minimal formal documentation to record the changes arising from close communication with clients.

The idea of introducing conceptual modeling into software development is to prevent developers from needing to understand massive text-based documentation. In software engineering, a conceptual model (a high-level abstraction of the real system)
represents the software system. Conceptual modeling plays a critical role in software
development. At least three groups of individuals are aware of the conceptual models.
First, analysts and designers carry out a requirements analysis and subsequently build
the conceptual model. Second, developers access the model to develop and verify the
needs of clients. Third, maintainers use the model to maintain and enhance system
functionalities. Various conceptual models serve as bridges between reality and
conceptual comprehension for all stakeholders during the entire software development
process (Kung 1989).

Models have levels of abstraction. Developers work closely with the most
concrete models that directly represent program code and communicate with customers
using the most abstract models that describe business requirements. For example, high-
abstract diagrams provide different views of the users’ needs and serve as efficient
communication tools between developers and end users (Larman 2004). Developers
could benefit from these diagrams by focusing on essential customer requirements and
monitoring requirement fulfillment by revising these diagrams at the beginning of each
iteration. Low-level abstract diagrams such as class diagrams and sequence diagrams
could be mapped to code directly. With the help of these diagrams, developers could
better understand existing programs and more quickly locate problems than they would
be able by reading the code directly (Larman 2004).

However, given the advantages of conceptual models, some practitioners argue
that excessive use of diagram documentation, such as UML diagrams, in an extreme
programming production process hinders rather than facilitates progress (Dori 2002).
Using diagrams introduces an extra cost to the development process. Extra efforts to
produce and understand these diagrams introduce additional overhead to the rapid
communication process. However, software development progresses through a series of
modeling changes, where each change incrementally refines the model. The conceptual modeling documentation represents a series of changes and is an efficient tool to handle the growing complexity of software development (Hungerford et al. 2004).

The literature confirms that conceptual modeling benefits software development and that diagrammatic models are superior to text-based models. A diagrammatic model is better in terms of software inspection (Hungerford et al. 2004), syntactic comprehension (Kung et al. 1995), semantic comprehension (Parsons 2002), schema-based problem solving (Shanks et al. 2002), and inferential problem solving (Bodart et al. 2001). In the stream of structured software development, entity-relationship diagrams (ERD) and data-flow diagrams (DFD) have been empirically proven to be efficient for the inspection software design defects (Hungerford et al. 2004). In the Object-Oriented (OO) software development paradigm, studies found that UML use case diagrams help developers understand the problem domain (Gemino et al. 2009).

In summary, the conflicts between the practitioner and the academic views of conceptual modeling are about whether the benefit of conceptual modeling justifies the extra cost it introduces. It is, therefore, necessary to investigate whether the use of diagrammatic representation makes a practically significant improvement that would justify the costs. In the context of software evolution, diagrammatic representation consumes most of the software development resources during the software development life cycle, and entails the comprehension of complex systems under constantly changing requirements.

One way to justify the extra cost is to use graphical models selectively. A software re-engineering tool automatically produces diagrammatic models (e.g., UML diagrams) of a complex system (Svensson et al. 2005). This automatic process introduces many diagrams that depict the system, but this process also requires extra
effort to understand and determine irrelevant or overlapping diagrams. Because none of the current studies have investigated how to select graphical models rigorously (Wand et al. 2002), an experimental study is needed to address the question of how diagrammatic tools contribute to comprehension and performance.

Distributed cognition theory (Hutchins 1995) provides a good conceptual foundation for addressing this question. Based on distributed cognition theory, one may argue that informational relevant artifacts in the environment have the potential to interact with the internal cognitions of developers to yield a richer perceptual system for reasoning and solving a problem (Zhang et al. 2007, Mangalaraj et al. 2014). Different types of UML diagrams provide various levels of external and internal representations that impact developers’ comprehension and problem-solving abilities through cognitive processes (Hungerford et al. 2004). The experiment designed and conducted in this study utilized individuals performing different software maintenance tasks. The experiment is expected to show salient performance differences that could suggest minimizing UML documentation by using only relevant diagrams. The results will lead to practical guidance for improving maintenance performance while maintaining documentation.

1.1 Importance of Research

Software systems have become more complex in terms of functionality and design. Research surveys estimate that there are more than 100 billion lines of code in production, 80 percent of which are unstructured and poorly documented (Talib et al. 2010). This creates consequential maintenance issues because the software systems become increasingly difficult to understand, particularly for new maintainers. It is, therefore, not surprising that maintenance costs are very high. Additional problems of outsourcing projects and maintainer turnover have made maintenance a problem for...
organizations. Lientz and Swanson surveyed 486 companies to investigate and compare application software maintenance cost. They reported that as much as 70 percent of all costs incurred on a software system are related to software maintenance (Lientz et al. 1980). Thus, it is essential to investigate which tools are appropriate for reducing maintenance efforts while improving performance.

Studies have found that conceptual models could reduce early design errors (Kung 1989) and save time and cost in later maintenance phases (Ramamoorthy et al. 1984). In the software analysis and design literature, conceptual models have been found to bridge customers and developers (Littlewood et al. 1989; Nissen et al. 1996; Ramesh et al. 1992). Conceptual modeling helps improve analysis and design performance by improving comprehension of the system. However, few studies have investigated whether the conceptual models are also useful during the software evolution phase (Arisholm et al. 2006). Normally the maintainers are not the original designers or developers of the software. Thus, maintainers must rely on the codes and extra documentation, such as system descriptions and diagrams, to help them understand the software. Therefore, the importance of conceptual models during the software evolution phase is no less than their importance during the analysis and design phases. Maintainers need the help of conceptual models to understand the system before they can perform changes. Investigating effective usage of conceptual modeling during the software evolution phase is necessary and beneficial. Investigating effective usage of conceptual modeling during the software evolution phase could reduce early design errors and lower costs in later maintenance phases by helping maintainers understand the software during various phases.

The lack of documentation becomes more pronounced as software development methodologies shift from traditional, plan-driven approach to Agile. Some basic software
development principles have changed as well (Nerur et al. 2007). For example, eXtreme program (XP), a popular Agile approach, interweaves coding with modeling and design and advocates the continual generation of self-documenting code through pair programming. One of the core principles is incremental, small release with frequent testing (Mangalaraj et al. 2009). In that sense, XP focuses on the solution space and tries to use programming techniques to obtain the simplest possible design as fast as possible, while the time and effort required to produce conceptual model documentation is limited to only what is absolutely necessary (Mangalaraj et al. 2009). Further, the emphasis on active customer/user participation is yet another distinguishing characteristic of agile. Therefore, introducing conceptual modeling is consistent with Agile methodology because it emphasizes communication between end users, stakeholders, and developers (Ambler 2002).

In reality, developers consider some forms of conceptual models necessary, but they rarely agree on what is needed (Briand 2003). In everyday practice, people use incomplete, obsolete documentation. Agile developers still need a conceptual model tool, but the tool should be adaptive to small and rapid changes without requiring much more time and effort. Hence, the primary goal of the present research is to investigate which conceptual model meets those needs.

To achieve this research goal, studies about the effectiveness of representation type must be reviewed. Codes and text-based system descriptions do not facilitate understanding of a large and complex software system. Relationships and logical processes are hidden in the text-based documentation (Amer 1993). By contrast, diagrammatic conceptual models such as entity-relationship models, data-flow models, and UML models present the system’s structural and behavioral aspects more explicitly (Brandimarte et al. 2000; Insfrun et al. 2002). Since these diagrams are widely used in
practice, this study will focus on diagrammatic UML models to investigate the aforementioned research issues.

The extant literature offers no conclusive findings on which type of UML diagram is superior to others. While developers use class diagrams, sequence diagrams, and use case diagrams most often (Dobing et al. 2006), experimental studies have found other diagrams useful as well. Collaboration diagram usage is related to higher comprehension levels than sequence diagrams in designing a real-time system. Sequence diagrams do not perform significantly better than collaboration diagrams on an MIS system (Glezer et al. 2005). State diagrams, as another dynamic modeling diagram, provide higher semantic comprehension of dynamic modeling when the domain is real time, and sequence diagrams are better in case of an MIS system (Otero et al. 2004b). These studies show that the effectiveness of UML diagrams depends on task type.

Motivated by the importance of conceptual modeling and the neglect of studies about software evolution performance, this research investigates the impact of the fit between type of conceptual modeling and task type on software evolution performance. The results would help developers select and use conceptual models for the right task, reduce efforts to handle extensive but irrelevant documentation, and focus on performing the task in a precise and efficient way.

1.2 Comparison to Existing Similar Studies

A few empirical studies have investigated the impact of conceptual models on comprehension and maintenance performance based on the task type (Agarwal et al. 1999; Khatri et al. 2006a; Shaft et al. 2006). Table 1-1 divides the current studies into four cells according to the context (program paradigm) and the tools (structural model or behavioral model). A valid comparison should be either across the columns (i.e., the structural models among different program paradigms) or the rows (i.e., both structural
and behavioral models within the OO program paradigm. This section summarizes these studies and discusses the differences between the present study and the existing ones based on the 2 x 2 classification in Table 1-1.

Agarwal et al. (1999) compared the developers’ comprehension, given the object-oriented (OO) model or the process-oriented (PO) model. They defined the PO model as a traditional structured technique and argued that the PO model was built around processes and tended to focus on the system’s behavioral or process-oriented aspects. The OO model, on the other hand, was relatively new in 1999, tending to emphasize both structural and behavioral aspects. Agarwal et al. used the theory of cognitive fit and posed two questions: “Is it easier to understand structural aspects of an application represented using an OO model rather than a PO model?” and “Is it easier to understand procedure aspects of an application represented using a PO model or an OO model?” Object class diagrams were used to generate OO models, and data-flow diagrams (DFDs) were used to produce PO models. An experiment was conducted using undergraduate students who were randomly assigned to one of the two models. Their comprehension of applications was measured afterward. Unfortunately, no salient differences were found between using the OO and PO models to understand the application’s structural or behavioral aspects. However, the PO model was found superior when the subjects were required to answer both structural and behavioral questions. A potential reason for the no salient results is the comparison between an object-oriented structural model (class diagram) and a traditional structured process model (DFDs). These two models are not comparable because they are used for different program paradigms and represent different aspects of the application. As shown in Table 1-1, the comparison is between cell 2 and cell 3. However, this paper is the initial inspiration for the present study because it defines the structural and behavioral model. Moreover, it is
an essential reference to design the comprehension measurement of structural and behavioral aspect of the application separately.

Table 1-1 Categories of Similar Studies

<table>
<thead>
<tr>
<th>Modeling Aspects</th>
<th>Program Paradigm</th>
<th>Structured</th>
<th>Object-Oriented</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Structural Model</td>
<td>Cell 1</td>
<td>Khatri et al. (2006)</td>
<td>Shaft and Vessey (2006)</td>
</tr>
<tr>
<td></td>
<td>Cell 3</td>
<td>Agarwal et al. (1999)</td>
<td>Present Study</td>
</tr>
<tr>
<td>Behavioral Model</td>
<td>Cell 2</td>
<td>Agarwal et al. (1999)</td>
<td>Cell 4 Present Study</td>
</tr>
</tbody>
</table>

Using the notion of cognitive fit, Khatri et al. (2006) investigated the impact of both IS and application domain knowledge on comprehension and problem-solving tasks. All subjects in the experiment had access to an entity-relationship (ER) model that either presented a familiar application domain or an unfamiliar one. The program used in this particular study was a structured design program and therefore falls into the top left quadrant in Table 1-1. Khatri et al. (2006) found that IS domain knowledge is essential in all comprehension tasks, but the effect of the application domain knowledge depends on comprehension type, i.e., semantic comprehension or schema-based problem-solving comprehension. The findings were significant and meaningful. However, the study discussed only the effect of the structural model (ER model) on comprehension. It is desired to revisit this in terms of both structural and behavioral models. In addition, Khatri et al.’s study (2006) study does not investigate whether the model plays any significant role in influencing coding performance.

The findings of Khatri et al. (2006) were expanded in a subsequent study that relied on cognitive theory to examine the relationship between software comprehension and maintenance (Shaft et al. 2006). It argues that a software maintainer’s understanding of the application domain tends to focus on software functionality. Therefore, if the
maintainer is proficient in the application domain and asked to perform a structure-oriented task, a cognitive fit exists and leads to better coding performance. By contrast, if
the maintainer is unfamiliar with the application domain, he or she should focus on understanding how the software accomplishes tasks, including control flow and state of change embedded in the code. Therefore, maintainers would have an advantage on behavioral modification tasks. However, this study has two limitations. First, this study was conducted in the context of COBOL programming, which is a structured programming language. An updated experiment should test whether the results are consistent in the context of OO programming. Second, the study uses a text-based conceptual model to depict the system. An updated study with a graphical conceptual model should be done to investigate the issue.

These existing studies focused on an important topic, i.e., the effectiveness of conceptual modeling when different types of tasks are undertaken. All these studies consider the task types in a similar way: structural oriented or behavioral (process) oriented. They also adopted cognitive fit theory to explain the relationship between conceptual models and task types.

Unfortunately, these existing studies have some limitations, and further investigation is desired. First, two of these studies focus on comprehension tasks, while only Khatri et al. (2006) investigate the effect of conceptual models on maintenance coding tasks. Future research should shift attention from comprehension to real coding performance. Second, these studies neglect the OO program paradigm. Instead of ER and DFD models, more OO models should be studied. UML is the most popular tool to conceptualize a domain for OO programming but has not been widely investigated in the context of task types.
1.3 Research Questions

This study's research goal is to investigate the impact of fit between various diagrammatic UML models and task type on software evolution performance. Specifically, the study attempts to examine how the interplay among different types of diagrams, tasks and comprehension impact performance. Therefore, the study addresses the following questions:

1. Does the choice of diagrammatical conceptual models affect software evolution performance?
2. Is the choice contingent upon the nature of programming task?
3. Does the programmer’s comprehension level mediate the association between the choices of conceptual models and software maintenance performance?

1.4 Overview

The remainder of the dissertation is organized as follows. Chapter 2 provides a literature review of relevant prior research as well as theoretical foundations that inform this study. From the perspective of IS technical domain, chapter 2 provides a review of software maintenance, conceptual modeling, and UML models. For the theoretical perspectives, a review of cognitive theory, distributed cognition theory, cognitive fit theory, and task-fit theory is provided. The research model is also developed at the end of chapter 2. Chapter 3 elucidates the hypotheses based on the research model and provides the theoretical justification for them. The experimental design, data collection, and analysis methods are discussed in chapter 4. Subsequently, the results are presented in chapter 5. Chapter 6 concludes the dissertation with a discussion of the implications of our findings for practitioners and IS academics.
Chapter 2

Literature Review

2.1 Software Maintenance

In this section, a thorough review of software maintenance is presented in 2.1.1. Potential research topics are then identified after the review. Section 2.1.2 and 2.1.3 explain two topics that are relevant to the current study.

2.1.1 Review of Software Maintenance

Software development activities can be classified into two categories: development and maintenance (Cote et al. 1990). Maintenance is different from development because developers have to spend a significant amount of time to understand the purpose and construction of the system (Banker et al. 1997). In the last two decades, the cost of maintaining software increased from 50 percent to 90 percent of the total cost of software development (Eastwood 1993; Erlikh 2000; Lientz et al. 1980; Port 1988). Due to the effort required for software maintenance, it could be a separate project from a new software development project. It is different from a system development project in that developers have to spend a significant amount of time understanding the program before making any modification (Banker et al. 1997).

However, the success rate of a maintenance project is relatively low. One survey shows that 23 percent of maintenance projects were canceled before full delivery, while only 28 percent were finished on time and budget with the expected functionalities (Seacord et al. 2003).

Researchers have investigated issues that may affect maintenance project success since the 1970s. At that time, the maintenance problem was aimed primarily at fixing bugs (McLean 1979). However, the area of software maintenance was largely neglected until the 1980s. The popularity was caused by a practical issue: the available
human resources for new development were restricted by the increasing number of systems combined with the growing volume of enhancement and maintenance (Lientz 1983). The maintenance tasks were not limited to fixing bugs anymore. A small survey of a hundred systems revealed that approximately 60 percent of the maintenance effort was for perfective maintenance, including system enhancement, documentation improvement, and coding optimization for efficiency (Lientz et al. 1978). For the first time, maintenance and enhancement tasks were found to be an important part of the software development life cycle, consuming approximately half of the system and programming personnel hours.

An extensive survey of two thousand professionals later showed five severe problems in software maintenance (Lientz 1983). First, the quality of application software documentation was poorly maintained. The documentation was too expensive to maintain with proper tools. Second, user demand for enhancements and extensions increased dramatically. Development professionals could not respond to those changes quickly and efficiently. Third, the time allocation between system maintenance and new product development was competing for programmers’ time. Fourth, developers felt it was difficult to meet scheduled commitments. Finally, the turnover in the development team left the system even more difficult to maintain and enhance.

Despite the fact that much of the research was conducted in the eighties, the findings are compatible with later software development studies. Schneildewind (1987) summarized the state of software maintenance in 1987: software development tools, methods, and measurement metrics are the three areas covered in the discussions of solving software maintenance problems. Accordingly, later researchers focused on finding 1) metrics, 2) factors, 3) efficient tools, and 4) methodologies to reduce the impact
of maintenance issues. Table 2-1 summarizes the studies in the last few decades regarding software maintenance in these four areas.

First, four studies built up measurement metric tools that measured the cost of maintenance tasks (Grady 1987; Kafura et al. 1987; Nguyen et al. 2011; Yau et al. 1985). The quality of the code was measured as the tool that controls maintenance work (Gibson et al. 1989). This type of study was popular in the eighties because the metrics were the foundation of all the other studies in this area. However, this topic was revisited as new software development methodologies emerged (Nguyen et al. 2011). Nguyen found that the measurement metric of object-oriented (OO) programming was different from older studies focusing on structured programming.

Second, eight out of twenty-five studies in Table 2-1 were about identifying important factors affecting maintenance task cost and performance. Tools were proposed to reduce the negative impact of these factors on maintenance cost and performance. One of the most well-studied factors is the complexity of the system (Banker et al. 1991; Banker et al. 1993; Banker et al. 1998). The complexity is caused by the increasing size and functionalities of information systems. It is critical to manage complexity where it cannot be eliminated. In fact, complexity is the greatest limiter in system maintenance. Survey studies reported that the estimated lines of programming code being maintained by organizations increased from 120 billion to 259 billion in the last decades (Sommerville 2000; Ulrich 1990). On average, a Fortune 100 company maintains 35 million lines of code with a 10 percent increasing rate each year only in enhancements (Muller et al. 1993).
<table>
<thead>
<tr>
<th>Reference</th>
<th>Analysis &amp; Design Methods</th>
<th>Language</th>
<th>Method</th>
<th>Interested Factors</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Yau et al. 1985)</td>
<td>-</td>
<td>-</td>
<td>Mathematical algorithm</td>
<td>Measurement Tool</td>
<td>Designed measurement metrics to reduce maintenance effort</td>
</tr>
<tr>
<td>(Bendifallah et al. 1987)</td>
<td>Structured</td>
<td>-</td>
<td>Case Study</td>
<td>Work Place Contingencies</td>
<td>Professional incentive encourages better maintenance work</td>
</tr>
<tr>
<td>(Grady 1987)</td>
<td>Structured</td>
<td>-</td>
<td>Survey</td>
<td>Measurement Tool</td>
<td>Designed measurements to improve software quality</td>
</tr>
<tr>
<td>(Kafura et al. 1987)</td>
<td>Structured</td>
<td>Database</td>
<td>Experiment</td>
<td>Measurement Tool</td>
<td>Designed complexity metrics to track changes and improve maintenance performance</td>
</tr>
<tr>
<td>(Gibson et al. 1989)</td>
<td>Structured</td>
<td>COBOL</td>
<td>Experiment</td>
<td>Quality of Code Structure</td>
<td>The more structured versions of the same software requires less time to maintain on average.</td>
</tr>
<tr>
<td>(Kung 1989)</td>
<td>Structured</td>
<td>Prulog</td>
<td>Experiment</td>
<td>Conceptual Model—ER &amp; DFD</td>
<td>Enable developers to understand users’ requirement better.</td>
</tr>
<tr>
<td>(Cordy et al. 1990)</td>
<td>Structured</td>
<td>Turing</td>
<td>Case Study</td>
<td>User Friendly Coding Environment</td>
<td>Good interface reveals both structural and nonstructural aspect of the system, facilitating maintenance task performance</td>
</tr>
<tr>
<td>(Podgurski et al. 1990)</td>
<td>Structured</td>
<td>Basic</td>
<td>Case Study</td>
<td>Control Flow Graph</td>
<td>Use control flow graph to conceptualize syntactic and semantic dependences, which facilitate software testing, debugging, and maintenance.</td>
</tr>
<tr>
<td>(Banker et al. 1991)</td>
<td>Structured</td>
<td>COBOL</td>
<td>Experiment</td>
<td>Software Complexity</td>
<td>Complexity influence maintenance cost through comprehension of the system. Proper usage of tools such as CASE has an impact on maintenance cost by reducing software complexity.</td>
</tr>
<tr>
<td>(Banker et al. 1993)</td>
<td>Structured</td>
<td>COBOL</td>
<td>Experiment</td>
<td>Software Complexity</td>
<td>Complexity influence maintenance cost through comprehension of the system. Proper usage of tools such as CASE has an impact on maintenance cost by reducing software complexity.</td>
</tr>
<tr>
<td>(Lejter et al. 1992)</td>
<td>OO</td>
<td>C++</td>
<td>Case Study</td>
<td>Inheritance Dynamic Binding</td>
<td>Proper Codes editor reduces the negative impact of complexity introduced by inheritance and dynamic binding.</td>
</tr>
<tr>
<td>(Wilde et al. 1992)</td>
<td>OO</td>
<td>C++</td>
<td>Case Study</td>
<td>Inheritance Dynamic Biding Polymorphism Structure disperse High abstraction (OO features )</td>
<td>Identify potential maintenance issues introduced by these OO features and Recommendations are made for possible tool support.</td>
</tr>
<tr>
<td>(Takang et al. 1996; Taylor et al. 1997)</td>
<td>Structured</td>
<td>COBOL</td>
<td>Case Study</td>
<td>Business Domain System Domain</td>
<td>The domains are both essential and complementary to the maintenance process.</td>
</tr>
<tr>
<td>Study References</td>
<td>Methodology</td>
<td>Technology</td>
<td>Type</td>
<td>Findings</td>
<td></td>
</tr>
<tr>
<td>------------------</td>
<td>-------------</td>
<td>------------</td>
<td>------</td>
<td>----------</td>
<td></td>
</tr>
<tr>
<td>(Banker et al. 1998)</td>
<td>Structured</td>
<td>COBOL Packaged Software</td>
<td>Experiment</td>
<td>Software Complexity</td>
<td>Software practices (code generator or packaged software) have impact on maintenance cost by influencing software complexity.</td>
</tr>
<tr>
<td>(Dishaw et al. 1998)</td>
<td>Structured</td>
<td>COBOL</td>
<td>Field Study</td>
<td>Task Fit</td>
<td>Fit between a maintenance task and the available maintenance tools is associated with maintenance support software tool use.</td>
</tr>
<tr>
<td>(Shaft et al. 1998)</td>
<td>Structured</td>
<td>COBOL</td>
<td>Verbal Protocol Analysis</td>
<td>Application Domain System Domain</td>
<td>Level of familiarity of application domain impact on comprehension strategy.</td>
</tr>
<tr>
<td>(Corritore et al. 1999)</td>
<td>Structured OO</td>
<td>C and C++</td>
<td>Experiment</td>
<td>Application Domain System Domain</td>
<td>OO expert acquired more application domain than procedures expert did to perform modification tasks.</td>
</tr>
<tr>
<td>(Ko et al. 2005)</td>
<td>OO</td>
<td>JAVA</td>
<td>Experiment</td>
<td>Usage of Eclipse IDE</td>
<td>Maintenance work consists of three fundamental activities focused on forming, navigating, and manipulating a working set of task-relevant code fragments.</td>
</tr>
<tr>
<td>(Lim et al. 2005)</td>
<td>Structured OO</td>
<td>C++ C</td>
<td>Field Study</td>
<td>OO or NOO Analysis and Design Usage of graphical models</td>
<td>Developers modified the OO version with help of UML diagrams required less effort to maintain software than their counterparts who worked on the NOO version.</td>
</tr>
<tr>
<td>(Souza et al. 2005)</td>
<td>Agile</td>
<td>Java</td>
<td>Survey</td>
<td>Various types of documentation</td>
<td>Importance ranking of documentation for software maintenance.</td>
</tr>
<tr>
<td>(Arisholm et al. 2006)</td>
<td>OO</td>
<td>Java</td>
<td>Experiment</td>
<td>Usage of graphical models</td>
<td>Improve functional correctness of changes as well as the quality of their design.</td>
</tr>
<tr>
<td>(Ko et al. 2006)</td>
<td>OO</td>
<td>Java</td>
<td>Experiment</td>
<td>Usage of Eclipse IDE</td>
<td>Design interface features to reduce time performing the mechanics of navigation within and between source files, and to help developers seek, relate, and collect information in a more effective and explicit manner.</td>
</tr>
<tr>
<td>(Shaft et al. 2006)</td>
<td>Structured</td>
<td>COBOL</td>
<td>Experiment</td>
<td>Task Fit Familiarity of Application domain VS Function or control flow Modification Task</td>
<td>Higher comprehension level leads to better performance only when fit exists between task type and application domain.</td>
</tr>
<tr>
<td>(Dzidek et al. 2008)</td>
<td>OO</td>
<td>Java</td>
<td>Experiment</td>
<td>Usage of graphical models</td>
<td>UML was beneficial in terms of functional correctness, but not quality or productivity.</td>
</tr>
<tr>
<td>(Nanthaamornphong et al. 2011)</td>
<td>OO</td>
<td>Java</td>
<td>Experiment</td>
<td>Usage of Design patterns</td>
<td>Did not improve either the maintainability or the understandability of the software.</td>
</tr>
<tr>
<td>(Nguyen et al. 2011)</td>
<td>OO</td>
<td>C++</td>
<td>Experiment</td>
<td>Measurement Tool</td>
<td>Designed measurement metrics to estimate maintenance effort.</td>
</tr>
</tbody>
</table>
Through software development phases, the complexity is measured not only by the numbers of lines of source code but also by many other characteristics of software (Banker et al. 1993). The lack of use of proper tools is believed to increase the cognitive load on software developers. Banker found that, in the context of structured programming paradigm, system complexity could be measured on three dimensions: procedure size, module size, and branching complexity. The study proposed that system complexity has a positive impact on software comprehension, which in turn influences the maintenance cost. Using proper tools, Computer Aided Software Engineering (CASE) to restructure code, will reduce the system complexity, improve comprehension, and reduce maintenance costs.

Experimental studies, such as Banker’s series of studies, identified complexity as the greatest limiter in software maintenance. They argued that the increasing complexity of systems was caused by the increasing size and incremental functional changes of information systems. The critical issue is to manage complexity where it cannot be eliminated. The lack of use of proper tools increases the cognitive load on software developers. Banker investigated the efficiency of several tools, such as CASE tool (Banker et al. 1993) and packaged software (Banker et al. 1997). The argument is that complexity influences maintenance cost through comprehension of the system. Proper choice of tools has an impact on maintenance cost by reducing software complexity. These arguments have been examined by experiments. However, all these experiments were conducted in the context of structured programming using coding language like COBOL. As the software development paradigm evolves to an OO paradigm, conducting similar studies is encouraged, but in terms of OO analysis and design using OO programming languages. In particular, the tools used to reduce OO programming
complexity are different from those used for structural programming. The usage of tools will be discussed in section 2.3.

The other well-studied factor is domain knowledge. The knowledge of the application (application domain knowledge) and the coding environment (system domain knowledge) are used differently under various circumstances. Shaft (1998) found that developers with different familiarity levels of the application domain knowledge would adopt different comprehension strategies. Programmers with a higher level of knowledge of the system domain require more application domain information than those with a lower level of system domain to perform modification tasks (Corritore et al. 1999). However, studies have found that application domain knowledge is important to both experts and novices. Application domain knowledge could reduce the comprehension differences between experts and novices (Parsons 2002; Villeneuve et al. 1997). These studies suggest that the notion of task fit is important. That is, the relationship between comprehension and maintenance performance is likely to be moderated by the fit between the familiarity of the application domain and the modification task. For example, Shaft et al. (2006) found that higher comprehension level leads to better performance only when fit exists between the task type and the application domain. Since most of these studies were conducted in the context of structured programming, it is important for us to affirm these findings in contemporary settings.

A survey of fifty-two developers from 237 maintenance projects revealed that source code, comments, graphical models, and user requirements descriptions are the top ranked documentation artifacts used (Souza et al. 2005). Experimental studies found that conceptual modeling diagrams enable developers to understand the application domain better during the maintenance phase (Arisholm et al. 2006; Dzidek et al. 2008; Kung 1989). However, the effectiveness of using documentation in the maintenance
phase is rarely studied and fully deserves thorough investigation (Hadar 2004; Hungerford et al. 2004; Kung 1989).

In addition to the aforementioned gaps in the extant literature, the investigation of these issues in the context of new software development methodologies - such as OO programming, particularly in an agile environment – is sparse. For traditional structural programming, it is not uncommon to break down the software system under development by functionality. Such an approach is inflexible in terms of handling requirements changes. By contrast, OO systems are relatively more modular, reusable, extensible and flexible. OO programming introduces new concepts, such as classes and objects, operator overloading, inheritance, dynamic/static binding, and powerful data abstraction facilities (Lejter et al. 1992; Wilde et al. 1993). A lack of understanding of OO concepts may interfere with developers' ability to understand code during the maintenance phase. For OO programming, developers need to understand not only the static aspect of the system but also the interactions among functionalities. The experimental study by Lim (2005) found that even though the developers perceived higher complexity of the OO system, they were able to modify the OO system faster than those who worked on the non-OO system. UML diagrams were found to aid OO developers (Arisholm et al. 2006; Dzidek et al. 2008). However, the benefits of UML diagrams were not conclusive, and more experimental studies are expected to investigate when and how to use UML diagrams in OO and Agile projects.

In summary, the review of existing studies about software maintenance revealed two general areas that deserve more attention. As OO, particularly in agile software development, becomes popular, a re-examination of some of these issues in the context of contemporary tools and techniques is required to confirm previous findings. The factors and tools that are important for maintaining structurally designed systems may not
influence the modern maintenance task in the same way. The benefits of documentation that are conferred on those who use modern development methodology have not been justified very well and require more research (Souza et al. 2005). As tools, techniques and methods evolve, it becomes important for us to ascertain how the fit of these tools and techniques with the task at hand affects outcomes.

With a view to extending prior research and also addressing some of the gaps in the literature, the current study uses a software maintenance task to investigate how tools, particularly conceptual modeling tools, influence perfective maintenance tasks in the context of an OO system. The fit between conceptual modeling tools and perfective maintenance tasks is also investigated in the current study. The following sections discuss perfective maintenance, and show how it differs from other types of tasks.

2.1.2 Perfective Maintenance

Software maintenance refers to any changes that must be made to software after delivery. The definition of software maintenance by IEEE is depicted as the process to correct errors, to improve performance, or to adapt the product to a modified environment after the software delivery (Engineers 1993). There are four types of maintenance tasks: corrective, adaptive, perfective, and preventive (Standardization 2006).

Among the four maintenance types, only corrective maintenance deals with malfunctions and defects. The other three types are generally fall under the purview of software evolution. The term evolution has been used to characterize the growth dynamics of software since the early 1960s (Chapin et al. 2001). Now the term is widely used in the software maintenance community. The Journal of Software Maintenance added the term evolution to its title to reflect this transition (Chapin et al. 2001).

Perfective maintenance is found to be a dominant part of the evolution process. It mainly deals with new or changed customer requirements. A successful piece of code
should be able to adapt to a succession of changes. It is based on the premise that if the software is useful, the users will experiment with new circumstances beyond the initial scope of the system (Takang et al. 1996). One study shows that about 65 percent of maintenance tasks are perfective tasks (Lientz et al. 1980). About 75 percent of maintenance costs are used for adaptive and perfective maintenance (Van Vliet 2000). To narrow down the topic and address the most critical issues, this dissertation investigates how to use graphical models to improve perfective maintenance.

2.1.3 Complexity Issues with OO Programming Language

When developing new systems, more and more organizations adopt object-oriented programming language instead of traditional procedural/structural programming languages (Wilde et al. 1992). The extensive reuse of software objects and efficient adaptation to changes through data encapsulation make OO programming the prevailing language in the industry now.

However, the statement “object orientation facilitates change” is based on the premise that developers have understood the system. Unfortunately, the concepts that advanced OO programming language also increased the complexity of the system. The increasing complexity causes a severe issue in perfective maintenance—“the more complex a system is, the more difficult it is to understand, and therefore to maintain” (Gibson et al. 1989). For traditional structural programming, it is common to break the software systems down by functionality. By contrast, OO programming increases system complexity by introducing new concepts, such as encapsulation, inheritance, dynamic binding, and powerful data abstraction (Lejter et al. 1992). When designing a system, OO developers will try to encapsulate the internal linkage of data structures and the details of the procedures into classes. However, the hiding structural and behavioral information can interfere with developers’ ability to understand the code and to locate the system
behavior they have to modify (Wilde et al. 1992). Therefore, the complexity introduced by OO programming to perfective maintenance tasks could be classified as structural complexity and behavioral complexity.

Structural complexity is caused by a new type of relationship introduced by the relationship between inheritance and polymorphism (Kung et al. 1995). In the traditional structured programming, the relationship between two entities captures how entities are related to one another, and the relationships are simply considered as verbs, linking two or more nouns (Chen 1975). In the OO programming, class is an exchangeable concept with entities in structure programming. However, the relationship between classes is further refined, basically including aggregation, composition, and generalization (Larman 2004). It is important to fully understand these relationships. Any changes to the attributes or operations in one class may have a ripple effect on all the other classes associated with it. While reading source code only, software maintainers have a hard time identifying the ripple effect of any changes intuitively because of the complex dependencies.

The behavioral complexity is caused by encapsulation (Wilde et al. 1992). Encapsulation is a unique concept to OO programming that makes source code more reusable. However, encapsulation forces the hiding of detailed procedure information in the code (Kung 1989). For example, when implementing an operation, it is hard to know the purpose and behavior detail of the operation. Only the name of the operation and the source of class are mentioned in the implementation statement. It could be easy for the developers to reuse pieces of codes during the development phase, but it would add a comprehension burden to maintainers when they need to change the code. If maintainers do not understand the purpose and behavior process of the operation, then they will not think to reuse the operation even if it is highly similar to the change requirements.
Therefore, a lack of understanding of the purpose of encapsulated operations diminishes the re-usage advantages brought by encapsulation and makes maintainers’ adaptation to the required changes less efficient.

Studies have found that the advantages of OO programming are prominent only when complexity is controlled and reduced by proper tools (Arisholm et al. 2006; Dzidek et al. 2008; Lim et al. 2005). However, the findings on the benefits of UML diagrams were not conclusive, and more experimental studies are required to investigate when and how to use UML diagrams in the context of maintenance of an OO system.

2.1.4 Documentation Issues with OO and Agile Methodologies

Developers from the analysis and design phase leave thousands of lines of code without explicit explanation for maintainers. The communication gap makes the software development process scattered, no matter what type of methodologies are adopted. Without a systematic understanding of the whole system, any improper changes to the system can bring in more severe issues than before (Seacord et al. 2003). Software engineers and researchers have been striving to find the best practices to smooth the development process and help maintenance. Documentation has been a prominent recommended practice for a long time (Ambler 2002; Anquetil et al. 2007; Takang et al. 1996; Tilley et al. 2003).

Documentation is an important tool to facilitate comprehension for both procedural programming languages and OO programming languages. However, documentation has not been accepted genuinely in practice. In a practical software project, it is not uncommon that software documentation is poor and incomplete. In order to cope with certain quality standards, most software projects require documentation. However, the documentation often conforms to a template and standards, but lacks
crucial information to make code understandable and usable by maintainers (Briand 2003).

Agile methodology is arguably the most dominant approach to software development. This approach places a premium on working code, while advocating that documentation be restricted to only what is absolutely necessary (Larman 2004; Mangalaraj et al. 2009). For example, eXtreme programming (XP), one of the more popular agile approaches, has provided different views from what has been assumed in conventional software development methodologies. If applied faithfully, developers should exert no efforts to produce unnecessary analysis and design documentation (Briand 2003). Incremental development in short iterations with frequent testing is one of the key characteristics of this approach (Mangalaraj et al. 2009). In that sense, XP focuses on the solution space and tries to use programming techniques to obtain simple design as fast as possible. Further, it advocates a collaborative environment in which customers play an active role, clarifying requirements and providing continual feedback. With this emphasis on working code, informal diagrams and documentation are deemed to be more helpful than formal analysis and design documents to record the requirements (Briand 2003).

From a long-term perspective, the lack of formal documentation in agile may come in the way of providing adequate design information to future maintainers to enable them to effectively maintain or evolve the system. In practice, a trade-off has to be made between the objective of producing working code of value in short iterations and the need for documentation that provides design traceability to make the maintainer’s job easier. With the preceding discussions in mind, the current study addresses the following question: How does one identify the most useful type of documentation for software
maintenance/evolution? Responses to this question are context-dependent, so the current study focuses its attention on Java and the Unified Modeling Language (UML).

2.1.5 Summary

In summary, the enormous cost and failure of software maintenance is caused by a variety of reasons. In order to manage software complexity and to smooth the software development process, developers look for efficient documentation that could improve the understanding of the system. Researchers found that conceptual modeling, as a concise and highly abstract document, is useful to reduce perceived code complexity (Kumar et al. 2004) and provides better communications among development groups (Kuechler et al. 2006). Conceptual models are concise and highly abstract documentation. Conceptual models improve the comprehension efficacy by modeling the domain, extracting high-level information from code, and creating abstractions to describe the system structure (Bolloju 2009; Brandimarte et al. 2000; Burton-Jones et al. 2009). A review of conceptual modeling is presented below.

2.2 Conceptual Modeling

Conceptual modeling is an approach capable of abstracting and representing various aspects of a domain (Kung 1989). It has been used in different areas to capture the semantics of the real world. The idea of introducing conceptual modeling into software development coincides closely with the idea of emphasizing user participation in the development of the requirements specification (Kung 1989). Models are better vehicles than source code to facilitate communication among users and developers, because conceptual models, such as entity-relationship diagrams, data flow diagrams, and UML diagrams, are usually in graphical form. It has been empirically verified that graphical representation is a better communication vehicle than textual representation (Burton-Jones et al. 2008; Lutters et al. 2007). Graphical conceptual models are used in
the area of software development to facilitate design and implementation processes by representing the semantics of the application domain as perceived by stakeholders of the information system (Burton-Jones et al. 2009). Producing conceptual models is an important step in developing an information system that gathers adequate user requirements about the application domain from structural and behavioral aspects. Failure to use these models could increase difficulty in identifying application domain knowledge and, subsequently, result in delayed scheduling, poor software/coding quality, and higher maintenance costs (Lin et al. 2004). The detailed relationship among conceptual models, understanding of the domain, and the performance of software development will be discussed separately in the following sections. The structural and behavioral aspects of the conceptual modeling are then discussed and further defined.

2.2.1 Conceptual Modeling and Comprehension

The relationship between conceptual modeling and application domain comprehension has been widely studied. The consistent finding is that conceptual modeling contributes to effective comprehension and communication (Burton-Jones et al. 2008; Lim et al. 2002). The level of effective comprehension and communication varies by knowledge of the application domain (Khatri et al. 2006a; Ricca et al. 2010; Shaft et al. 1998). There are several characteristics of conceptual modeling that impact developers’ comprehension ability, including modeling notations, modeling approaches/tools, and modeling content (Wand et al. 2002). The impact of each characteristic on comprehension is reviewed separately in the following sections.

2.2.1.1 Modeling Notations

A large body of conceptual modeling research argues how to make conceptual models easy to understand by studying the semantics of modeling constructs based on ontology theory (Hadar et al. 2006). The efficacy of different representations, such as
notations and styles, has been studied. For instance, research has found that using stereotyped notation in conceptual models reduces the gaps between subjects with less skill or experience and highly skilled or experienced subjects (Ricca et al. 2010). Optional attributes and relationship notations are useful to developers only when the developers need a surface understanding of the system (Bodart et al. 2001). The detailed level of notations also influences the developers’ comprehension level (Nugroho 2009). However, this is not in the scope of this study.

2.2.1.2 Modeling Approaches and Tools

The choice of conceptual modeling approach is another key factor that affects developers’ comprehension ability. Generally, conceptual modeling can be developed by structure-driven/procedural or object-oriented approaches (Lin et al. 2004). The choice of design paradigms and developing languages decides the usage of different modeling approaches. For example, the structured programming paradigm usually adopts a top-down conceptual model, which maps the domain into separated sub-domains. Each sub-domain is coded in a separate module and can only be accessed and read through a single point. Models like the entity-relationship model and data-flow diagrams are produced to conceptualize the application domain in the structured programming paradigm, which is influenced by languages like COBOL and C. The limitations of such an approach become increasingly obvious as systems grow more complex. Some of the shortcomings of the structured paradigm, such as difficulties in understanding, maintenance and reuse, are overcome to a degree by object-oriented concepts such as inheritance, encapsulation and polymorphism (Lim et al. 2005). The Unified Modeling Language (UML) has become a standard for modeling the application domain in an object-oriented manner and the representation resulting therefrom may be implemented in object-oriented languages such as C++ and Java.
The languages that is used to implement the model often determines the modeling approaches. The effectiveness of modeling tools was investigated under certain situations. In the last decade, the advantages of using structural modeling tools, such as entity-relationship model and data-flow model, to improve domain knowledge have been confirmed in the structural programming paradigm. Process model was found to be easier to understand than the OO model when the system was designed by structure-oriented methodology (Agarwal et al. 1999). Further, Agarwal et al.’s study demonstrated that a data-flow diagram is easier to understand than an object diagram when developers have little knowledge about the structural aspects of the domain. The deficiencies of structural modeling tools are also found empirically in the context of OO programming paradigm (Kung 1989; Kung et al. 1995; Podgurski et al. 1990). The consistent finding is that structural modeling tools, including entity-relationship and data-flow models, are inadequate for stakeholders to understand an OO-designed application domain (Kabeli et al. 2005; Lim et al. 2005). By contrast, the effectiveness of OO modeling tools has not been studied as thoroughly as the structural modeling tools. Although UML diagrams, such as class diagram, sequence diagram, and communication diagram, have been found in separate studies to help OO developers greatly (Arisholm et al. 2006; Dzidek et al. 2008), this finding is not always conclusive (Dzidek et al. 2008). When all-UML diagrams were adopted, UML diagrams were only found empirically beneficial in terms of functional correctness but not quality or productivity because developers spent too much time switching between diagrams to gain understanding (Dzidek et al. 2008). There are no conclusive findings of the effects of UML diagrams (Hungerford et al. 2005), and more experimental studies are needed to investigate when and how to use UML diagrams for developing, maintaining and evolving object-oriented systems. The present study uses UML models as the tool to investigate the issues of conceptual modeling in OO software.
development. Instead of using a single UML diagram or adopting all-UML diagrams in one study, the current study investigates which UML diagram is superior to others and should be used mainly for different software development tasks. A review of UML diagrams in section 2.3 will provide a better understanding of modeling tools discussed in this paper.

2.2.1.3 Modeling Context

The conceptual modeling context is the situation under which conceptual modeling occurs. Therefore, besides the characteristics of the conceptual model discussed in the last section (notation, methods, and tools), studies about how the contextual factors affect modeling work deserve further attention (Wand et al. 2002). Two contextual factors have been examined: individual factors and task factors.

Individual contextual factors, such as expert versus novice and individual cognitive characteristics, have been found to moderate the relationship between comprehension and solution performance (Wand et al. 2002). Studies involving novices versus experts found that the experts take more time to develop a holistic understanding of requirements (Villeneuve et al. 1997). Expert also produce high-quality solutions in terms of accuracy, completeness, innovativeness, and adaptability. However, the gap between experts and novices is filled when comprehension (as internal representation) fits the task representation (Khatri et al. 2006b; Shanks et al. 2008). In the present experiment, subjects from similar programming classes are randomly assigned to each group.

Task contextual factors like task type have been studied. Task type could be defined based on task characteristics; for example, simple versus complex (Shaw 1954) and easy versus difficult (Bass et al. 1958). Conceptual modeling is advocated to reduce the impact of task complexity and difficulty (Lejter et al. 1992). For example, research
results suggest that 3D graphs outperform 2D graphs for both complex and simple tasks (Kumar et al. 2004). However, we are not interested in the complexity of a task in this study. It is controlled by assigning the same task to subjects within the same investigation group in the present study.

Literature on software development also defines task type in terms of the structural and behavioral nature of the application domain being studied. A structural model telling what the system is and a behavioral model telling how the system performs together constitute the representations of a system (Kung 1989). As the software system becomes complex, the system cannot be represented as a structural or behavioral system alone. Some systems may be more structural in nature, but may contain interactions and dynamics that may be represented using a state machine. For example, while a database system is more about data structure and relationships, it also contains retrieval and other routine behaviors (Amer 1993). By contrast, a collaboration information system focuses on process and behaviors to allocate resources (Lin et al. 2004). Design of a real-time system also emphasizes process and interactions rather than static structures (Glezer et al. 2005).

Even the application domain cannot be categorized as a pure structural system or a pure behavioral system, as studies investigating the differences between structural-oriented and behavioral-oriented systems have found them to exhibit both characteristics. When working on a structure-oriented system as opposed to a process-oriented system, programming performance is more sensitive to the application domain knowledge level (Shaft et al. 2006). The fit between the structural/behavior task and representation style impacts the comprehension level of the task (Agarwal et al. 1999). Therefore, the present study adopts the classification of structure-oriented task and behavior-oriented task. To be concise and maintain consistency with the previous study, the terms \textit{structural task}
and behavioral task are used throughout the present study instead of “structure-oriented task” and “behavioral-oriented task.”

2.2.1.4 Summary of implications

In summary, the present study draws on two streams of conceptual modeling literature. Specifically, it investigates whether structural and behavioral object-oriented (OO) models play different roles in comprehension. The study investigates UML models because it is the de facto standard for both behavioral and structural OO modeling. The investigation is conducted in the context of different types of tasks, namely, structural- and behavior-oriented.

2.2.2 Conceptual Modeling and Performance

While many studies have examined the relationship between conceptual modeling and comprehension (Burton-Jones et al. 2008; Kumar et al. 2004; Nugroho 2009; Ricca et al. 2010; Tilley et al. 2003), very few have investigated the direct relationship between conceptual modeling and task performance (Arisholm et al. 2006).

Previous research has explored the nature of this relationship using two phases of problem solving, namely, comprehension and problem solving (Shaft et al. 2006). These studies have largely relied on cognitive theories (see discussions in section 2.4.1). However, there are many contextual factors that could adversely affect the impact of external and internal representation on final performance, and these have not been studied systematically using experimental design. Thus, there is a need to for a methodologically rigorous and theoretically grounded experiment that can provide valuable insight into the relationship between modeling and performance. In particular, it is important to understand how the fit between the task and the nature of the conceptual model used impacts outcomes.
2.2.3 Conceptual Modeling Representation

Conceptual modeling relies on visualization to help developers quickly understand the system. In general, the models used by developers are graphical and involve notations that are fairly easy to understand.

2.2.3.1 Graphical Modeling

Diagrams are normally high-level abstractions of the real system. Previous researchers have performed extensive studies comparing graphical models to textual models in terms of software inspection (Hungerford et al. 2004), syntactic comprehension (Kim et al. 1995), semantic comprehension (Parsons 2002), schema-based problem solving (Shanks et al. 2002), and inferential problem solving (Bodart et al. 2001).

The general findings are consistent with human cognitive theory. Cognitive theory indicates that when information enters the human mind, it is processed in sequential stages in short-term memory. Additional information will be retrieved from long-term memory first, and then the results of cognitive processing are organized and saved in long-term memory (Barber 1988). When developing a complex information system, diagrams are often superior to text-based representations because diagrams reduce cognitive load by shifting information acquisition load to the visual perception system. In the structured software development literature, there is empirical evidence to support the hypothesis that entity-relationship diagram (ERD) and data-flow diagram (DFD) are efficient tools with which to inspect software design defects (Hungerford et al. 2004). In the stream of OO software development paradigm, findings suggest that UML use case diagrams are useful representations to help developers understand the problem domain (Gemino et al. 2009). The majority of studies have examined only single-diagram usage during the software development process, while in reality multiple diagrams are often used simultaneously. There is a paucity of studies that have
investigated the integration effect of multiple diagrams (Kim et al. 2000). The experimental findings indicate that as the system becomes more complex, users need multiple diagrams to represent it. Multiple diagrams depict a system from different perspectives and at different levels of abstraction. For example, different functional roles may require different ways of conceptualizing the problem, and, therefore, my need diagrams that can support varying levels of abstraction. Previous researchers have argued that UML diagrams - the de facto standards in object-oriented development - model a system in terms of its static and dynamic characteristics (e.g., Booch et al. 19XX). Modeling the static and behavioral aspects of a system creates a thorough understanding of the problem domain (Pennington 1987).

As the software development process becomes more agile to customer requirement changes, practitioners prefer to use minimalistic documentation during the software development process (Nerur et al. 2005). The best practices of Agile methodology encourage developers to document only what is absolutely necessary and nothing more. Agilists believe that the code should be self-documenting (see Beck 2004) and tend to use UML diagrams sparingly, often sketching quick diagrams (e.g., class diagrams or sequence diagrams) before proceeding to code. This reluctance to use elaborate diagrams may be a source of concern to some as there seems to be some empirical evidence to suggest that UML diagrams are effective representations of the real world (Kim et al. 2000). However, we need more empirical affirmation of the efficacy of static and dynamic diagrams to convince the practitioner community that diagramming can only enhance and not diminish the effectiveness of the software development process. In particular, it is important to understand how and under what circumstances these diagrams can be of value.
2.2.3.2 Structural and Behavioral Aspects

Conceptual models abstract many different aspects of reality (Wand et al. 2002). Insfrun (2007) suggested that there are two types of conceptual modeling: one to illustrate static aspects of the system and one to describe the dynamics of the system under development (Insfrun et al. 2002). However, the OO programming languages introduce additional concepts that encapsulate the structure (e.g., class attributes) and behavior (e.g., instance methods) of the system. It makes the differences between the structural and behavioral aspects more salient than the differences between dynamic and static aspects of an application domain (Parsons et al. 2008). Therefore, the current use of different models to abstract structural and behavioral aspects of an application domain remains separate.

The structural model represents the static aspect of the system, including classes and their attributes as well as the structural relationships among them (Booch et al. 1999). The behavioral conceptual model represents the dynamic aspects of the system, including interactions among the objects and/or the various states through which an object belonging to a particular class might transition (e.g., state machine or state chart diagram). The interaction is a behavior that comprises a number of messages exchanged among a set of objects within a particular circumstance to perform a specific task (Booch et al. 1999).

The integrated understanding of both structural and behavioral aspects of a system is highly required in OO analysis and design practices (Snoeck et al. 1998). It is important to have a comprehensive view of the system at the beginning. Popular UML diagrams, including class diagram (Kabeli et al. 2005), sequence diagram (Erickson et al. 2007), and object diagram (Agarwal et al. 1999), were found wanting in these experimental studies because they don’t integrate both structure and behavior in a single
diagram. However, when the software development process moves to the maintenance phase, maintainers only need to understand the issues they are working with, instead of all the details of the system. Therefore, the goal of this study is not to test the overall usefulness of UML diagrams, or to test single diagram effectiveness, but to compare the effectiveness among UML diagrams and find the right UML diagram to use under certain task conditions.

In addition, communication diagram is a type of UML diagram, which to an extent integrates structural and behavioral aspects in the same diagram. Unfortunately, this diagram is largely ignored by existing studies because of it is not very popular in practice (Grossman et al. 2005). The present study investigates the effectiveness of this integrated diagram and two other diagrams—class diagram for structural aspect and sequence diagram for behavioral aspect. Specifically, the study examines how the three UML diagrams influence the maintainer’s comprehension of the system and how that, in turn, impacts performance.

2.3 UML Diagrams

Unified Modeling Language (UML) is an accepted standard tool for the analysis and design of OO systems. Management has also used UML as a tool to support software maintenance. However, companies don’t always use UML diagrams as was intended. Users must develop many UML diagrams to comprehensively represent a single system because each type of UML diagram represents a different aspect of the application domain. Because of their large size and complexity, UML diagrams are commonly criticized as being hard for users to comprehend precisely and comprehensively (Dzidek et al. 2008). More empirical studies are needed to investigate the benefits of using UML in realistic contexts. Perhaps, the first thing to examine is the complexity of UML, which is usually discussed in terms of the number of objects,
relationship among objects, and property per technique (Erickson et al. 2007). However, due to UML’s complexity, the myriad variations of object, relationship, and property representations make it difficult to achieve any conclusive answers about how to control the complexity.

An alternative to control the complexity of UML is to use UML diagrams selectively. The duplications among different types of UML diagrams waste valuable resources (Gelbard et al. 2010). There may also be overlaps among the diagrams. For example, both sequence diagrams and activity diagrams represent the same functional procedures and involved objects, and sequence diagrams and the communication diagrams represent the same functional procedures, albeit the former emphasizes the timing of messages while that latter draws attention to the structural relationships as well. We, therefore, argue that there is no need to adopt all UML diagrams while building software. Rather, developers can benefit more by selecting diagrams based on how well they fit the task. This could save them time and effort, and leave more time to comprehend the diagrams that truly matter depending on nature of the task at hand. To our knowledge, no study explores the selective strategies of UML diagrams according to the nature of the task. A realistic question is waiting to be answered: Given a certain task, which diagram could provide the most reliable semantic information about the system? The answer to this question could help the software industry make better decisions about which UML diagrams they should adopt in development practices.

To address the above question, the first step is to categorize UML diagrams into structural and behavioral diagrams (see section 2.3.1). There are 3 diagrams of interest in this study, namely, class diagrams, sequence diagrams, and communication diagrams. The unique features of each diagram are discussed in sections 2.3.2 to 2.3.4.
In section 2.3.5, the existing studies about UML diagrams are reviewed, and research opportunities are identified.

2.3.1 Structure and Behavior

Insfrun (2007) suggests that there are two types of conceptual modeling, one to illustrate static aspects of the system and the other to describe the dynamic elements of the system (Insfrun et al. 2002). However, the OO programming languages introduce the additional concepts that encapsulate structure (data) and behavior (method) of the system. OO designers use UML diagrams to abstract out the structural and behavioral aspects of the system being modeled. According to Booch (1999), major UML diagrams are generally classified as structural and behavioral diagrams. Class diagrams and object diagrams are structural diagrams, while use case diagrams, sequence diagrams, collaboration diagrams, state chart diagrams, and activity diagrams are behavioral ones (Larman 2004). Structural diagrams identify the system, and behavioral diagrams identify the system’s function. Software design, test, and maintenance are fundamentally concerned with both the structure (what it is) and the behavior (what it does) of the system under development.

Structural diagrams emphasize what must be modeled in the system. They show a collection of static model elements, such as classes, their relationship, and content. Main components include classes, responsibilities, associations, dependencies, inheritance relationships, composition associations, and association classes. Behavioral diagrams, on the other hand, emphasize how the system works. They include sequence diagrams, collaboration diagrams, state chart diagrams, and activity diagrams (Larman 2004). While use case diagrams address the static view of a system, sequence diagrams, collaboration diagrams, state chart diagrams, and activity diagrams all address...
the dynamic view of a system. All of these diagrams are useful in organizing and modeling the behaviors of a system.

The integrated understanding of both structural and behavioral aspects of a system is highly required in OO analysis and design practices (Snoeck et al. 1998). It is important to have a comprehensive view of the system at the beginning. The question is: do developers benefit from using UML diagrams at the maintenance phase? It is our contention that UML diagrams can help maintenance programmers by providing a high-level perspective of the underlying code, thereby enabling them to quickly navigate the solution space in search of an appropriate solution. However, not all diagrams may be useful to evolve a solution to the problem at hand. In particular, the nature of the task may dictate the type of diagram one should use.

Despite the large number of UML diagrams, the current study will focus on only one structural diagram (class diagram) and two behavioral diagrams (sequence and communication diagrams). The class diagram and sequence diagram are included in the present study because they are the two diagrams on the extreme ends of representing structural or behavioral aspects of the system. The class diagram abstracts the static structure of the system, while the sequence diagram represents the behavioral aspects of the system, including dynamic interactions and sequence of messages (Otero et al. 2004b). The communication diagram provides both structural and behavioral information, but less than the information that the class diagram and sequence diagram together provide (Larman 2004).

Additional reasons not to cover all-UML diagrams are discussed in the UML literature section. The review of class diagrams, sequence diagrams, and communication diagrams is presented in the following sections.
2.3.2 Class Diagram

The class diagram describes the static features of the software model. It shows classes with operations and attributes connected by different kinds of static relationships or associations. It also has notations that can enrich the semantic representation of the domain being modeled. For example, whole-part relationships – also known as aggregation – can be distinguished from a regular association. Furthermore, whole-part relationships in which the part cannot have an independent existence (i.e., a composition relationship in which the part belongs to only one whole) can also be easily modeled (Booch et al. 1999). Class diagrams also enable modelers to represent generalization and specialization, often referred to as inheritance. The specifications of the class diagram are listed in Table 2-2. With these notations, the class diagram shows how abstract data types, such as people, things, devices, items, and the like, are related to one another.

Table 2-2 Notations of Class Diagram (Srivastava, 2009)
Figure 2-1 is an example of a class diagram, which is also used in this study’s experiment. The business and personal classes inherit from the customer class. The association notation between the account class and customer class indicates that each customer object is associated with multiple account objects in the source code (but not the other way around). The account class is also associated with the transaction class.

The advantage of using a UML class diagram becomes apparent when change requests for a system are similar to existing structure during the maintenance phase. Using the system represented in Figure 2-1 as an example, if the new requirement is to add a savings and checking account to this system, maintainers can easily extend the Account class to create Savings and Checking Account classes that inherit the attributes and methods of the Account class. This structure has some advantages. One may extend the sub-classes without modifying the code in the client that uses the inheritance structure. Furthermore, one can avoid duplicating methods implementations that are common to sub-classes.

It is also more productive to locate the lines of code that require changes. For example, suppose the system needs to change the transaction display content. After reading the class diagram, it is easy to notice that the display is embodied in the toString() method in the Transaction class. Therefore, maintainers could modify the appropriate method in the Transaction class to accommodate the required display changes.
Figure 2-1 Example of Class diagram: Banking System
2.3.3 Sequence Diagram

Sequence diagrams simply model interactions between objects represented in a time sequence. The sequence diagrams cannot represent the overall structure of the system as the class diagram does. However, they highlight the sequence of interactions among objects in a given scenario. The sequence of messages shows the details of how the system behaves in order to accomplish its functionality. However, since each sequence diagram represents object interactions for a use case scenario (i.e., one pass through the use case), users have to read multiple diagrams to gain a complete understanding of the system works. The elements of sequence diagrams are shown in Table 2-3.

Table 2-3 Notations of Sequence Diagram (Srivastava, 2009)

<table>
<thead>
<tr>
<th>Component</th>
<th>Description</th>
<th>Model Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object Lifeline</td>
<td>It represents the existence of an object at a particular time.</td>
<td>![Object1]</td>
</tr>
<tr>
<td>Activation</td>
<td>It shows the time period during which an object or actor is performing action</td>
<td>![Message1]</td>
</tr>
<tr>
<td>Message Call</td>
<td>A communication between objects that conveys information and results in an action</td>
<td>![Message2]</td>
</tr>
<tr>
<td>Return message</td>
<td>The return result of the message call</td>
<td>![Message2]</td>
</tr>
</tbody>
</table>

The example of a sequence diagram shown in Figure 2-2 represents the deposit transaction use case. This diagram is also used in our experimental study. From this single sequence diagram, the user cannot tell whether or not there is an inheritance relationship, which is an important structural aspect of the system. However, it clearly shows the objects (and therefore the classes to which they belong) involved in carrying
out a use case scenario. This understanding can be useful in adding another transaction that may involve some of the same objects. For example, if the requirement is to add the withdrawal transaction to the system, the easiest way is to copy all the processes involved in Figure 2-2 and modify them to create the withdrawal transaction shown in Figure 2-3.

In addition, it is also productive to make changes to similar procedures in a batch. For example, by reading multiple sequence diagrams, it is easy to identify the similarities between Figure 2-2 and Figure 2-3. If a test found any logic errors in the deposit transaction procedure, then it is reasonable to suspect the same errors in the withdrawal transaction procedure. Therefore, if the maintainers can recognize similar procedures from a sequence diagram, they can improve their productivity by fixing the same issues in similar procedures at the source code level.

Figure 2-2 Example of Sequence Diagram: Deposit Transaction
2.3.4 Communication Diagram

Communication diagrams and sequence diagrams are interchangeable. The elements of a collaboration diagram are shown in Table 2-4. Communication diagrams are similar to sequence diagrams because both diagrams represent interactions among objects involved in a use case scenario. The sequence diagram emphasizes only the chronological order of messages, whereas the collaboration diagram emphasizes the structural organization of the objects while showing the sequencing of messages. Even though the communication diagram cannot provide as comprehensive a structural view of the system as the class diagram, it can abstract the structural information to an extent by placing all elements in a hierarchal way.

Figure 2-3 Example of Sequence Diagram: Withdrawal Transaction
Table 2-4 Notations of Communication Diagram (Srivastava, 2009)

<table>
<thead>
<tr>
<th>Component</th>
<th>Description</th>
<th>Model Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object</td>
<td>It represents an instance of a class.</td>
<td></td>
</tr>
<tr>
<td>Association</td>
<td>It shows the relationship between object and the order in which they are access.</td>
<td></td>
</tr>
</tbody>
</table>

Figure 2-4 is an example of a communication diagram of the deposit transaction procedure. This diagram is also used in the experiment and is interchangeable with Figure 2-2. However, in Figure 2-4, the association relationship between the account object and transaction object is emphasized by the link between the two objects. This example is too small to exhibit the advantages of the communication diagram over the sequence diagram. When the diagram gets more complex, the association relationship among objects will be hard to detect in a sequence diagram but much easier to notice in a communication diagram because of the way the diagram places the elements. However, the communication diagram cannot show the multiplicities directly, but the developers can use the communication diagram to locate the lines of code that define the multiplicities. Therefore, communication diagrams have the characteristics of both class diagrams and sequence diagrams and can be useful in providing developers a glimpse of the structural and behavioral elements of a system involved in a given scenario.
The usefulness of UML diagrams has been widely studied. The consistent finding is that compared to text descriptions and source code, UML diagrams contribute to effective comprehension and communication (Burton-Jones et al. 2009; Lim et al. 2005).

However, there are no conclusive findings about which UML diagram is superior to others. While developers use class diagrams, sequence diagrams, and use case diagrams most often, experimental studies also find other diagrams to be of value. The collaboration diagram, as a conceptual model that models both the static and dynamic aspects of the application, achieves better comprehension for real-time systems than sequence diagrams, while sequence diagrams do not perform significantly better than collaboration diagrams for an MIS system (Glezer et al. 2005). The state diagram, another type of dynamic modeling UML diagram, provides higher semantic comprehension when the domain is real time, and sequence diagrams are better in an MIS system (Otero et al. 2004b). These studies show that UML diagrams are useful depending on the task.
Another stream of studies focuses on the domain knowledge of the developer. During the design phase, the availability of domain knowledge helps the developer achieve complete diagram design without reducing comprehension (Shaft et al. 1995). The theory of cognitive fit also plays a critical role in these studies. The findings indicate that IS knowledge is an important independent variable, while application domain knowledge has a contingent impact on performance (Khatri et al. 2006a). Cognitive fit exists when a developer with high-level application domain knowledge must perform a functional task (Shaft et al. 2006). The findings mentioned above indicate that when developers have high-level domain knowledge, they can adopt structure-oriented diagrams. On the other hand, when developers have low-level domain knowledge, they can adopt data flow-oriented diagrams to achieve similar comprehension performance.

The comprehension of UML diagrams during software development has been studied extensively. It is measured by free recall (Pennington 1987) or responses to comprehension questions (Corritore et al. 1999). As the name implies, free recall requires a developer to recall a program. It requires memorization, which may or may not involve deep understanding, and is used for only small programs. Comprehension questions frequently have been employed to assess a maintainer’s mental representation of software (Shaft et al. 2006).

Comprehension is essential but is only part of the maintenance task. During the maintenance phase, developers must understand the system so they can adapt to any modification changes, understand current code, correct any errors, and reuse and leverage code (Von Mayrbauser et al. 1995). Comprehension is only the basic requirement to perform all the tasks during the maintenance phase. To perform these tasks better, developers must have a holistic but efficient understanding of the system.
Despite widespread interest in UML, there are few quantitative studies about the level of UML usage and the interplay between the diagram and the task. The current study examines three UML diagrams used during the software maintenance phase and addresses two key questions: 1) To what extent do these three UML diagrams help improve comprehension and coding performance? 2) Do comprehension and the performance that ensues depend on the interaction between the task factor and the type of diagrams? This study relies on the following theoretical perspectives to address the questions of interest.

2.4 Theoretical Perspectives

Theories related to cognition process and cognitive fit help researchers interpret critical issues in the conceptual modeling literature. The following sections review three important theories related to conceptual modeling research and discuss how these theories relate to the present study.

2.4.1 Cognitive Theories

Cognitive theories are a group of theories that investigate how people understand the material and solve problems (Barber 1988). Cognitive theories suggest that cognition is a complex process. When information enters a person’s mind, the person processes that information in a series of ordered stages in short-term memory. Additional information may be retrieved from long-term memory, and the results of cognitive processing are revealed and subsequently saved in long-term memory (Barber 1988).

In software engineering, a conceptual model represents the software system. Conceptual modeling plays a critical role in software development. At least three groups of individuals are concerned with conceptual models. First, analysts, designers, and developers carry out a requirements analysis and subsequently build the conceptual
model. Second, the model(s) can be used to communicate with the users with a view to affirming the requirements of the system. Third, maintainers use the model to maintain and/or enhance system functionalities. Modeling helps the analyst, designer, and maintainers construct a conceptual representation of the system before programming begins. The conceptual representation permits the analyst, designer, and maintainers to gain full understanding of the system and helps prevent significant problems during actual implementation (Kung 1989).

Brooks (1983) developed a cognitive–behavioral theory of program comprehension in software engineering. He defines three types of domains that are main components of the problem-solving process. The first is the problem domain, which contains task descriptions waiting to be solved. The second is the solution domain, which contains the solutions to the tasks in the problem domain. When a programmer fully understands a program, what he/she knows can be described as a succession of knowledge domains that bridge the problem and solution domains (Brooks 1983). In terms of software perfective maintenance, a conceptual model is a high-level representation of the programs.

The cognitive process consists of perceptual and conceptual processes (Larkin et al. 1987). The perceptual process is a bottom-up activity that involves understanding the system structure, relationship, and process, while the conceptual process is a top-down activity that involves generating and refining solutions (Larkin et al. 1987). Studies have adopted cognitive process theory to investigate searching strategies among conceptual models (Kim et al. 2000; Kung 1989). The findings suggest that the right searching strategies among conceptual models will facilitate cognitive processes, thereby improving understanding of the system.
Therefore, cognition is a complex process that could be improved by using the right conceptual model. Conceptual models can help build incremental knowledge that fills the gap between the problem and solution domains. In light of cognitive process theories, software development researchers have extensively studied how conceptual models impact system comprehension. Notations, methods, tools, and contents of the conceptual models have different impacts on system comprehension (Wand et al. 2002). The evolution of conceptual model notations and grammar has been studied to provide fundamental but efficient support for conceptual modeling (Burton-Jones et al. 2009; Evermann et al. 2006). Conceptual models are used to describe a system's structural or behavioral aspects. Structural models, like the entity-relationship diagram and the class diagram, are proficient at representing a system’s structural and static aspects, while behavioral models like the sequence diagram and the flow chart are good at representing dynamic and behavioral aspect of the system (Bollen 2006; Bolloju 2009). Studies also investigate the impact of conceptual models on software inspection (Hungerford et al. 2004), syntactic comprehension (Kim et al. 2000), semantic comprehension (Parsons 2002), schema-based problem solving (Shanks et al. 2002), and inferential problem solving (Bodart et al. 2001).

These findings are consistent with cognition theories because the findings recognize the process of searching for solutions as two steps: 1) gaining understanding of the system and 2) delivering the solution. These findings are summarized in Figure 2-5, which is the basis for the final research model. In this study, the UML diagram is used as a particular tool to produce conceptual models of information systems.
2.4.2 Distributed Cognition Theory

Distributed cognition theory has provided the new foundation for human–computer interaction (Hollan 2000). The insight is that cognitive behaviors are best understood as distributed processes. Cognitive process is distributed across people and artifacts, and it depends on both internal and external representations (Zhang et al. 1994).

The theory is appropriate because traditional cognitive theories have many critics. The traditional cognitive theories emphasize an internalism that marginalizes the role of external representation and problem solving. These theories consider cognitive process to be a closed process with single agent or artifact (Hutchins 1995). It is also the reason that the majority of current researchers focus on only a single diagram each time, because the theory ignored the social interaction among developers and multiple diagrams. By contrast, the theory of distributed cognition considers the cognition process as a distributed process instead of a closed one. Information is collected through multiple distributed external representations (e.g., UML multiple diagrams). Therefore, from the distributed cognitive perspective, the cognitive process is an interactive search process among distributed external presentations (Zhang et al. 1994).

In the context of software development, developers must understand the problem domain by searching for useful information among distributed conceptual models. At the beginning, developers may believe a problem statement is useful and then turn to the use
case diagram because it is more intuitive. However, they may then return to the problem statement to find basic descriptions. Previous studies find that such an interactive process does exist between text-based representation and the use case diagram (Hungerford et al. 2004).

UML is a language used to describe reality at different levels of abstraction and from different perspectives (Booch et al. 1999). Therefore, it is an external representation to individual developers. According to Zhang (1994), experimental subjects made less errors with the help of external representations. External representations simplify complicated problems because experimental subjects can check external rules directly without overloading their working memory.

However, when there are too many external representations, individuals must use working memory to save external representations and understand them by connecting representations through cues. This indicates that external representations cannot improve understanding without limit. External representations can overwhelm an individual inspecting them directly by overloading the individual’s working memory and deteriorating the cognition process.

Zhang (1994) argues that external representations can change the task’s nature. For example, presenting the same problem with different representations (such as color, size, and shape) results in different perceptions of the task complexity because some external representations are internalized easily to reduce the perceived task complexity.

Distributed cognition theory suggests that different conceptual representations could also change the perceived nature of the comprehension task. Given the same system, using structural diagrams will easily internalize the system’s structural aspects, so users would gain more structural knowledge about the system. On the other hand, using behavioral diagrams will internalize behavioral aspects of the system, so users
would gain more behavioral knowledge about the system. Therefore, Figure 2-6 proposes the evolved research model.

UML diagrams, as the tools to produce the conceptual models, can be classified as structural diagrams and behavioral diagrams. Class diagrams are used in this study to represent the system’s structural aspects, while sequence diagrams are used to convey behavioral information. Communication diagrams are considered more complex tools that include both behavioral and structural information.

![Figure 2-6 Process to Increase Program Performance—Distributed Task](image)

### 2.4.3 Cognitive Fit Theory

Cognition literature indicates that the conceptual model can improve solution domain by providing incremental knowledge of the system to fill the gap between the problem and solution domains. The conceptual model’s availability significantly improves the functional correctness of changes and the quality of their design (Arisholm et al. 2006). However, incremental knowledge of the problem domain does not guarantee the best solutions, or the best solutions do not require complete knowledge of the problem domain. Maintainers often successfully modify software without understanding it fully (Corritore et al. 1999). Given the size of a software system, maintainers would not devote the time to develop a complete understanding of the entire problem domain while modifying the content of software development (Shaft et al. 2006). Thus, the relationship
among conceptual modeling, comprehension, and solution performance is more complicated than the direct relationship, as presented in Figure 2-5 and Figure 2-6.

Unfortunately, studies investigating the effectiveness of conceptual models through the lens of cognitive theories stopped at the comprehension phase without further investigating the relationship between comprehension and solution performance (Shaft et al. 2006). There are two reasons. First, conceptual modeling in the context of software analysis and design has been studied extensively, while little attention has been given to studies in the context of software maintenance. During the analysis and design phase, researchers are more interested in how effectively conceptual modeling represents the problem domain because the main task is to understand the requirements and better design the system (Glezer et al. 2005; Kumar et al. 2004; Otero et al. 2004b). The only thing of research interest in the software design and analysis literature is comprehension of the system because that comprehension is the output of the analysis and design phase (Kung 1989). Better comprehension may improve coding performance during the implementation phase or reduce the maintenance effort during the maintenance phase, but there is no need to investigate coding performance during the analysis and design phase. By contrast, the solutions of the modification task are the final output during the maintenance phase. Despite the importance of comprehension and its relevance to performance, few studies have made an effort to elucidate the relationship(s). This study fills this void.

The second reason is the complexities of the conceptual model context, which was discussed in section 2.2.1. Two contextual factors have been examined: individual factors and task factors. The controlled experiment procedure will reduce the effect of individual factors to a minimum. In addition, assigning the same task to subjects within the same investigation group controls the task’s complexity.
The contingent role of task type on the relationship between usage of tool and task performance is of research interest. Two mainstream cognitive theories have been adopted to explain this complicated issue.

Cognitive fit theory proposes that the correspondence between task and external information presentation format leads to superior task performance for individual users. In several studies, cognitive fit theory explains performance differences among users across different external presentation formats such as tables, graphs, and schematic faces (Shaft et al. 1995). The research also found that cognitive fit exists between internal representation of the task and task type (Shaft et al. 2006). Shaft (2006) found that if the fit exists between the developer’s comprehension of the task (internal representation) and the external representation of the application domain, improved comprehension would lead to better modification performance.

As discussed in section 2.5.2, there are two broad categories of diagrams – static and dynamic. These two types of models produce different types of external representations for the system: structural representation and behavioral representation. Within the same amount of time, developers would understand the corresponding aspect of a system faster and more efficiently by focusing on only one type of external representation of the system than they would by using all types of external representations (Briand 2003). If the task requires a lot of understanding of the system’s structure, using the structural model will help developers focus on the information useful to them. The time and effort would be saved from reading and understanding all these external representations. This argument also holds for the behavioral external representations. Shaft et al. (2006) supports this argument by rigorously defining cognitive fit as “the degree to which the information representation and any tools or aids employed match the information required to perform a task.” Therefore, the model in
Figure 2-7 emphasizes the importance of fit between the external representations (the models) and the task type.

### 2.4.4 Task-Technology Fit Theory

Task-Technology fit (TTF) theory is another important fit theory that has been developed particularly for information system research. The cognitive fit theory emphasizes the importance of the comprehension (the internal representation) that is an important antecedent to performance, while TTF is much more straightforward. It argues that any technique tool can only have positive impact on performance if the tool fits the task that it supports (Goodhue 2006).

Goodhue and Thompson (2006) define the fit as the "correspondence between task requirements and the functionality of the technology." The definition of fit is similar to Vessey’s (2006) but focuses more on the direct impact on performance. Goodhue and Thompson (2006) argue that a tool can affect task performance directly. If the task-technique fit exists, a tool could positively affect task performance by changing the possible execution sequence (Goodhue 2006).

For example, if the task requires changes to a system’s structure, a structural model could help developers locate the codes that need modifications. Developers would not spend much time understanding the system but would aim to solve the issue.
precisely and quickly. By contrast, if the developers use a behavioral model that provides details of interactions and processes, the developers would not find useful structural information from the model and would have to look at the code. In addition, the source code will not separate the structural information from the behavioral information, so developers would have to memorize all the system’s relationships and interactions even though they do not relate to the task. All these extra activities hinder the developers’ ability to perform a task. The fit between task type and model type has a direct positive impact on performance. Therefore, the research model in Figure 2-8 has a direct link from the FIT to the task performance.

![Figure 2-8 Process to Increase Program Performance—Direct Effect of FIT](image)

### 2.5 Research Model

The final research model is based on Figure 2-8 but further divided into two models according to the task type (Figure 2-9 and Error! Reference source not found.) for better experimental design.

Figure 2-9 represents the fit for a structural task. When performing a structural task, users can reference the structural diagram to build a static understanding of the system. A static understanding benefits the programming performance only when the
task is structural. Under this circumstance, the fit only exists between structural task and structural model. This fit is called a structural fit, or SFIT in the following discussion.

Similarly, if the task were behavior oriented, the user’s structural understanding would not be enough to perform any changes. Behavioral diagrams help the user build a dynamic understanding of the system. That dynamic understanding would benefit the programming performance in the context of a behavioral task. The fit under this circumstance is called a behavioral fit, or BFIT in the following discussion. The next chapter will develop hypotheses based on these two models.
Chapter 3
Hypotheses Development

3.1 Refined Research Questions

Zhang’s (1997) statement below states the core idea of the present study: the form of the representation matters.

“…external representations are not simply inputs and stimuli to the internal mind; rather, they are so intrinsic to many cognitive tasks that they guide, constrain, and even determine cognitive behavior.”

Zhang (1997) argued that information presented by external representations could be gained and processed by subjects without many interruptions from internal representations. Different forms of external representations could cause dramatically different cognitive behaviors of subjects under various circumstances. Some forms of external representation would facilitate the cognitive task, while some forms would hinder it. Conceptual modeling is a type of external representation that is designed to facilitate cognitive tasks such as reasoning, comprehension, problem solving, and decision making (Aguirre-Urreta et al. 2008; Khatri et al. 2006a; Khatri et al. 2006b). However, the effectiveness of the conceptual model depends on the form of the model and the nature of the task (Zhang et al. 1994). For example, it has been empirically verified that graphical representation is a better communication vehicle than textual representation (Burton-Jones et al. 2008; Lutters et al. 2007). Khatri et al. (2006) found that when a software maintainer with proficient application knowledge is asked to perform a structural-oriented task, then cognitive fit exists and leads to better coding performance. Therefore, the general research question in this study is to investigate the effects of different conceptual models on software maintenance performance and to determine the
contingency role of task types. In order to answer this question, a few terms have to be specified.

First, the present study uses UML diagrams as a tool to generate conceptual models due to their popularity in the software industry. As external representations, UML models are generally classified into structural diagrams and behavioral diagrams (Booch et al. 1999). Class diagrams and object diagrams are structural diagrams, while use case diagrams, sequence diagrams, collaboration diagrams, state chart diagrams, and activity diagrams are behavioral ones. Structural diagrams tell developers what the system is, and behavioral diagrams tell developers what the system does. Software design, testing, and maintenance are fundamentally concerned with structure (what it is) and behavior (what it does). Class diagrams, sequence diagrams, and communication diagrams will represent the structural diagram model, the behavioral diagram model, and the structural/behavioral diagram model.

Second, the task type has to be carefully defined. Task type could be defined based on the characteristics of the task; for example, simple versus complex (Dishaw et al. 1998) and easy versus difficult (Bass et al. 1958). Literature about software development also defines task type based on the structure of a task—the structural or behavioral nature of the application domain being studied. As the software system becomes huge and complex, not a single system can be depicted as a purely structural or purely behavioral system. Some systems are structurally complicated but still contain interactions and state machines. For example, a database system is about data structure and relationships, but it also contains retrieval and other routines representing behavior (Amer 1993). By contrast, a collaboration information system focuses on the process and behaviors to allocate resources (Lin et al. 2004). Design of a real-time system also emphasizes process and interactions rather than static structures (Glezer et al. 2005).
The present study adopts the classification of structure-oriented task and behavior-oriented task. To remain concise and consistent with the extant literature, the terms *structural task* and *behavioral task* are used throughout the present study instead of structure-oriented task and behavior-oriented task.

Third, cognitive fit theory and task-technology fit theory, discussed in section 2.4, suggest studying the fit between conceptual models and task type. To perform a task, maintainers need to process information perceived from conceptual models and the requirements of the task in an interwoven and dynamic way. The information that is perceived directly from the model constrains the range of possible cognitive actions in the sense that some actions are feasible, while others are hard to achieve (Zhang 1997). For example, given the class diagrams, the maintainers would easily gain knowledge about how to change the structural aspects of the system because the fit exists. On the other hand, if they were required to modify any process and behavioral aspects of the system, they would need more time and effort to accomplish the task because the model did not provide direct and explicit information regarding processes and behavior. Therefore, fit is defined as the degree to which a technology could assist a maintainer in performing tasks. The fit has twofold structural fit (SFIT) or behavioral fit (BFIT), depending on the task type. Each fit contains three levels: fit, non-fit, and partial fit, depending on the UML diagrams adopted. Table 3-1 summarizes the fit levels. When performing a structural task, maintainers assigned with a class diagram would find more relevant information to perform the task than using sequence or communication diagrams. The communication diagrams would provide some structural information but not as much as a class diagram. Therefore, structural fit exists between class diagram and structural tasks but does not exist between sequence diagram and structural tasks. However, partial fit exists between communication diagram and structural task. When performing a behavioral task, the
maintainers' structural understanding would not be enough to perform the changes. Behavioral diagrams help users build a dynamic understanding of the system. Such dynamic understanding of the system would benefit the programming performance in the context of behavioral tasks. Similarly, behavioral fit exists between behavioral models (sequence or communication diagrams) and behavioral tasks but does not exist between class diagrams and the behavioral tasks.

### Table 3-1 Cognitive Fit

<table>
<thead>
<tr>
<th>Structural Task</th>
<th>Structural Model (Class Diagram)</th>
<th>Behavioral Model (Sequence Diagram)</th>
<th>Behavioral Model with Partial Structure Information (Communication Diagram)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Structural FIT</td>
<td>Non-FIT</td>
<td>Partial Structural FIT</td>
</tr>
<tr>
<td>Behavioral Task</td>
<td>Non-FIT</td>
<td>Behavioral FIT</td>
<td>Partial Behavioral FIT</td>
</tr>
</tbody>
</table>

The general research question is divided into three specific questions listed below:

1. Does structural fit or behavioral fit result in better subject comprehension than non-fit or partial fit?
2. Does structural fit or behavioral fit result in better subject modification performance than non-fit or partial fit?
3. Does the comprehension level fully mediate the relationship between fit and modification performance?

The following section develops the hypotheses to answer these three questions.

### 3.2 Effect of Fit on Comprehension

**Research Question 1**

Does structural fit or behavioral fit result in better subject comprehension than non-fit or partial fit?
As external representations, UML models are generally classified into structural and behavioral diagrams (Booch et al. 1999). Class diagrams and object diagrams are structural diagrams, while use case diagrams, sequence diagrams, collaboration diagrams, state chart diagrams, and activity diagrams are behavioral diagrams. Structural diagrams tell developers what the system is and behavioral diagrams tell developers what the system does. Software design, testing, and maintenance are fundamentally concerned with structure (what it is) and behavior (what it does).

However, there are no conclusive findings that specify which type of UML diagram is superior to others. While developers often use class diagrams, sequence diagrams, and use case diagrams, experimental studies also found the usefulness of other diagrams. Collaboration diagrams, as conceptual models representing both static and dynamic aspects of the application, have been proven to achieve better comprehension than sequence diagrams for real-time systems. Sequence diagrams do not perform significantly better than collaboration diagrams for MIS systems (Glezer et al. 2005). State diagrams, as dynamic modeling diagrams, are found to provide higher semantic comprehension of dynamic modeling in UML in case of a real-time system, while sequence diagrams are better in case of a MIS system (Otero et al. 2004a). The inconsistent findings indicate that the overall comprehension is not a very good measurement. Certain types of diagram may only increase partial overall comprehension. Therefore, the overall comprehension has been divided into two separate parts in the present study: structural comprehension and behavioral comprehension. These two parts of comprehension will be tested separately.

The distributed cognition theory, discussed in section 2.5.3, is applied to explain how UML diagrams partially benefit comprehension. Zhang (1994) argued that external representations could change the nature of the task. For example, the same problem
depicted with different representations (such as color, size, and shape) results in different perceptions of the task difficulties. This is because in certain circumstances, some external representations are easier than others to internalize. This internalization process reduces the perceived task complexity and helps users to better perform the task (Zhang et al. 1994). Some conceptual models can be easily internalized into structural information, while others can be internalized into behavioral information. Whether the internalized information is beneficial to the task depends on the type of task.

The benefits of using models to gain comprehension are salient when the information provided by these models is also the information required to perform the task. Burton (2008) draws on the cognitive economy effect to propose that individuals would read multiple forms of representation only if they perceived it to be a worthwhile investment. If an individual already had enough information to solve the problem then he/she would stop using other forms of representation (Burton-Jones et al. 2008). Therefore, when maintainers are required to comprehend a structural task, those using structural diagrams would internalize the structural aspect of the system better than others using behavioral diagrams. Maintainers using structural diagrams would feel satisfied to answer all structural comprehension questions with the information perceived from the structural models and thus stop searching for alternative solutions. This situation occurs when structural fit exists that provides adequate structural information that is useful to the maintainer.

By contrast, if maintainers use behavioral diagrams to perform structural tasks, they would perceive less adequate information from these diagrams to guide their answering of structural questions. Before answering the comprehension questions, maintainers would have to spend more time inferring structural information from behavioral diagrams or, if necessary, look into source code to gain knowledge of the
system. Maintainers’ process of extra searching for useful information would impact their working efficiency and quality. This situation occurs when non-structural fit exists.

To be specific, class diagram is a structural diagram that shows the objects and relationships. With the help of the class diagram, a developer will gain an understanding of structural aspects of the system, including attributes of objects and specifications of operations. Relationships including association, dependency, aggregation, and composition among objects are also easy to retrieve from class diagram. In addition, class diagram is also an efficient artifact to represent a hierarchical relationship between super- and sub classes. Polymorphism could be learned if the users understand UML diagram notations. However, it would be very challenging to obtain deep structural information such as relationships and the inheritance structure from a behavioral diagram like a sequence diagram. If the maintainers have enough time, they may be able to infer structural information from code comprehension. However, their process of extra searching for useful information would detract from their working efficiency and quality. Therefore, the hypotheses are developed as the following:

\[ H_{1a}: \text{In a structural programming task, the use of class diagram will result in a better structural comprehension score than the use of sequence diagram, on average.} \]

\[ H_{1b}: \text{In a structural programming task, the use of class diagram will result in a better structural comprehension score than the use of communication diagram, on average.} \]

The communication diagram is defined as a behavioral diagram in UML and is interchangeable with the sequence diagram. It also contains more external information than sequence diagram. With the help of a communication diagram, developers will gain the benefit of not only a sequence diagram but also information about association
relationship among objects. However, the magnitude of the structural benefits may not be as large as the benefits gained directly from class diagrams. Developers will not obtain complex relationship information such as dependency, inheritance, etc. Therefore, the level of structural comprehension gained from the communication diagram is less than the level of structural comprehension gained from the class diagram but it is greater than that gained from the sequence diagram. In other words, the structural fit exists partially when the maintainer adopts communication diagrams to perform a structural-oriented task. An additional hypothesis is developed, as the following depicts the relationship between sequence diagrams and communication diagrams:

H1c: In a structural programming task, the use of communication diagram will result in a better structural comprehension score than the use of sequence diagram, on average.

The above three hypotheses are developed under the assumption that the programming task is structure-oriented. Similar hypotheses have also been developed for behavior-oriented tasks. If the maintainer works on a behavioral task, the maintainer using behavioral diagrams would internalize the dynamic and processes aspects of the system easily. Their understanding of the system would be behavior-oriented, emphasizing the processes and state machines. The behavioral fit exists and provides the adequate behavioral information that is more useful to the maintainers. The perceived information from behavioral diagrams would be good enough to answer the comprehension questions of a behavioral task. By contrast, if the maintainers adopt structural diagrams, the behavioral fit does not exist. Therefore, when performing a behavioral-oriented software task, less behavioral information would be transferred and perceived by the maintainers using structural diagrams than when using behavioral diagrams. In other words, sequence diagrams and communication diagrams are
behavioral diagrams that show the interaction and state machines. With the help of
sequence or communication diagrams, developers will gain understanding of the order of
message among multiple objects, which is hard to interpret directly from a class diagram.
In order to gain the knowledge of behavioral aspects of the system, the maintainers
would most likely search for answers in the source code. Source code is comprehensive,
but it is not a good representation of the domain. A good representation should provide
adequate information that can be perceived and used by users directly (Zhang 1997).
Reading the source to gain knowledge would only detract from maintainers’ working
efficiency and quality. Therefore, the second set of hypotheses is:

\[ H2a: \text{In a behavioral programming task, the use of sequence diagram will result} \]
\[ \text{in a better behavioral comprehension score than the use of class diagram, on} \]
\[ \text{average.} \]

\[ H2b: \text{In a behavioral programming task, the use of communication diagram will} \]
\[ \text{result in a better behavioral comprehension score than the use of class diagram,} \]
\[ \text{on average.} \]

Although the communication diagram is defined as a behavioral diagram in UML
and is interchangeable with sequence diagram, it does not show the order of messages
in the same fashion as a sequence diagram. Even with numbered sequence, the order
information from the communication diagram is not as intuitive as that from the sequence
diagram. Therefore, the level of behavioral comprehension gained from communication
diagram is less than that gained from sequence diagram, but it is greater than that gained
from class diagram. The last hypothesis in the first set is as follows:

\[ H2c: \text{In a behavioral programming task, the use of sequence diagram will result} \]
\[ \text{in a better behavioral comprehension score than the use of communication} \]
\[ \text{diagram, on average.} \]
3.3 Effect of Fit on Coding Quality

Research Question 2

Does structural fit or behavioral fit result in better subject modification performance than non-fit or partial fit?

To answer this question, task-technology fit theory is adopted. While the cognitive fit theory emphasizes the importance of comprehension (the internal representation) as an antecedent to performance, the task-technology fit is much more straightforward. It argues that a technique can only have a positive impact on performance if the tool fits the task type (Goodhue 2006). Task fit theory proposes that the correspondence between task and information presentation format leads to superior task performance for individual users. The underlying reason for the superior task performance is that proper fit provides accurate and adequate information for the maintainers to perform a task.

Given a structural task, developers assigned with structure diagrams would perform better than those assigned with behavioral diagrams because developers with structural diagrams build understanding in the same static format of the task. Given a behavioral task, developers with behavioral diagrams should perform better than the developers assigned with structural diagrams because developers with behavioral diagrams build understanding in the same dynamic format of the task.

The argument is also in the same light of Zhang and Norman’s study in 1994. They argued that it is not necessary to construct internal representations to mediate cognitive action. Individuals can perceive direct and instructive information from proper external representations and act upon it in an adaptive manner (Zhang et al. 1994). These cognitive actions may or may not result in good performance depending on whether the form of external representation fits the requirement of the task performed.
Structural fit exists when maintainers are required to enhance a structural programming task with the help of structural conceptual models. An UML class diagram is a specific example of a structural conceptual model. The advantage of using UML class diagrams becomes notable when change requests for a system are similar to existing structure during the maintenance phase. A simple copy-and-paste process and minor modification would do the job. It would save maintainers a lot of time spent recalling the inheritance grammar and key words and typing redundant codes. It is also more productive to locate the lines of codes where changes are required. Once the maintainers identify where to make the changes, they would transfer their attention from diagrams to source code to perform the task.

By contrast, the non-fit between structural task and behavioral diagrams would obstruct enhancement behaviors. To be specific, the sequence diagram is a type of behavioral diagram. From the sequence diagrams, the user cannot tell whether there is an inheritance relationship, which is an important structural aspect of the system. The maintainers using behavioral diagrams would find inadequate information from the diagrams to fulfill enhancement requirements for a structural task. They would spend more time and effort to explore more information, which would impair the efficiency and quality of work (Burton-Jones et al. 2008).

Therefore, the third set of hypotheses regarding the coding quality is presented as following:

\textit{H3a: In a structural programming task, the use of class diagram will result in better coding quality than the use of sequence diagram, on average.}

\textit{H3b: In a structural programming task, the use of class diagram will result in better coding quality than the use of communication diagram, on average.}
Hypotheses $H3a$ and $H3b$ are developed under the assumption that the undertaken task is a structural programming task. Similar hypotheses are also developed as follows for behavioral programming tasks.

Behavioral fit exists when maintainers are required to enhance a behavioral program task with the help of behavioral conceptual models. The UML sequence diagram is a specific example of a behavioral conceptual model showing the interaction and state machine. Sequence diagrams model simple interactions between objects represented in a time sequence. The sequence diagram cannot represent the overall structure of the system as the class diagram does. However, it displays the time sequence of objects participating in each function. The sequence of messages shows how the system behaves at a detailed level. Using the sequence diagrams, maintainers would be more productive to add new processes that are similar to the existing process. Simple copy-and-paste processes and minor modifications would do the job. In addition, it is also productive to make changes to similar procedures in batch. For example, by reading multiple sequence diagrams, it is easy to identify the similarities among procedures. If a test found any logic error in one procedure, then it is reasonable to expect the same error to occur in all similar procedures. Therefore, if the maintainers can recognize similar procedures from sequence diagrams, they can improve their productivity by repairing the same issues in similar procedures with a single visit to the source code.

Maintainers using class diagrams would not have the benefits discussed above. It is hard for maintainers to gain the information of the interactions among multiple objects. Without the information, maintainers would spend more time exploring source code. Source code is comprehensive, but it is not a good visual representation of the domain. A good representation should provide adequate information that can be perceived and used by users directly (Zhang 1997). Exploring the source to find the place
to make the changes and to write code from scratch would only detract from maintainers’ working efficiency and quality. Therefore, the fourth set of hypotheses is:

*H4a: In a behavioral programming task, the use of sequence diagram will result in better coding quality than the use of class diagram, on average.*

*H4b: In a behavioral programming task, the use of communication diagram will result in better coding quality than the use of class diagram, on average.*

The performance difference between maintainers using sequence diagrams and those using communication diagrams poses another set of interesting questions. The maintainer using communication diagrams during the cognitive process gains comprehension of both the structural and behavioral aspects of the system. Communication diagram is similar to sequence diagram in the way it represents each procedure in a separate diagram. Sequence diagram only emphasizes the time order of messages, while collaboration diagram captures both the structural organization of objects and the sequence of messages among them. When the system grows bigger, the association relationship among objects will be difficult to detect in a sequence diagram but much easier to notice in a communication diagram because of the way communication diagram places elements. However, the communication diagram could not confirm some structural aspects of the system directly, such as multiplicities. The developers can locate the lines of code where multiplicities are defined using the communication diagram but extra effort and time are required to do so.

Therefore, the communication diagram includes the advantages of the sequence diagram and some of the advantages of a class diagram. However, the magnitude of benefits may not be as large as the benefits derived directly from class and sequence diagrams. Even though structural and behavioral fits exist with communication diagrams they are weaker than the fits when class and sequence diagrams are used separately.
(Table 3-1). Hence, the question is whether the two weak fits are better than a single strong fit. As the software system becomes complex and huge, not a single system could be depicted as a purely structural or behavioral system. Some systems are of structural complexity but still contain interactions and state machines. For example, a database system is more about data structure and relationships (Amer 1993). A database system also contains retrieval and other behavioral routines, but it is less complex than structural. In addition, the behavioral and structural aspects of the system are tightly linked. Hence, when a customer requires improved system function, both structural and behavioral changes to the system are required. Multiple diagram studies also found that the developer will bounce back and forth among structured representation and behavior representation, while the search pattern depends on the nature of the task (Kim et al. 2000). Therefore, we would argue that even a communication diagram does not provide as strong structural or behavioral comprehension separately, as a class diagram or a sequence diagram does; it provides a more comprehensive view of the system than class and sequence diagrams, which benefits both structural and behavioral task outcomes.

The corresponding hypotheses are developed as follows:

\[ H3c: \text{In a structural programming task, the use of communication diagram will result in better coding quality than the use of sequence diagram, on average.} \]

\[ H4c: \text{In a behavioral programming task, the use of communication diagram will result in better coding quality than the use of sequence diagram, on average.} \]

3.4 Mediation Role of Comprehension

Question 3:

Does the comprehension level fully mediate the relationship between the fit and modification performance?
Sections 3.2 and 3.3 developed hypotheses about the direct effect of fit on comprehension and coding performance, separately. This part links together these two sets of hypotheses. The cognitive process theory (Wand et al. 2002) and existing studies (Bollen 2006; Bolloju 2009) consistently suggest that the process of searching for solutions consists of two steps: gaining understanding of the system first and then delivering the corresponding solution. As argued in sections 3.2 and 3.3, the overall comprehension may not benefit performance, but structural or behavioral comprehension may play different roles in accomplishing various types of tasks. When working on a structural programming task, maintainers using class diagram or communication diagram gain more structural comprehension than those using a sequence diagram. The higher structural comprehension level leads to better coding performance. Similarly, when working on a behavioral programming task, maintainers using communication diagram or sequence diagram achieve better coding quality than those using a class diagram because of the higher level of structural comprehension. In other words, proper fit improves coding performance through enhancing corresponding comprehension level. The last set of hypotheses is developed as follows:

\[ H3a: \text{In a structural programming task, structural comprehension mediates the impact of SFIT on coding quality.} \]

\[ H3b: \text{In a behavioral programming task, behavioral comprehension mediates the impact of BFIT on coding quality.} \]

However, according to the task-technology fit theory, this mediation relationship is not expected to be a fully mediated one. Goodhue and Thompson (2006) argued that there are many alternative ways to improve performance by task-technology fit. Improving comprehension through task-technology fit is only one of these alternatives. If the task-technique fit exists, a tool could also positively affect task performance by
changing the possible execution sequence (Goodhue 2006). For example, if the task requires changes to the structure of a system, a structural model could help developers locate the place where code modifications are needed. Developers would not spend much time understanding the system but would aim to solve the issue precisely and quickly. By contrast, if the developers use the behavioral model that provides details of interactions and processes, developers would not find useful structural information from the model, and then they would have to look into the code for detailed information. In addition, the source code will not separate the structural information from the behavioral information. So developers would have to memorize all the relationships and interactions of the system even though some of those may be unrelated to the task. All these extra activities obstruct developers’ ability to perform a task. Therefore, it is expected to find these two hypotheses supported partially in the experiment.

3.5 Summary of Hypotheses

Fourteen hypotheses were developed in the last section. These hypotheses were developed for structural programming tasks and behavioral programming tasks separately, which makes it easy to design two separate experiments in the next chapter. The general comparison is between structural diagram (class diagram) and behavioral diagram (sequence diagram and communication diagram). Hypotheses are also developed to test the difference between the two behavioral diagrams (sequence diagram and communication diagram). The fourteen hypotheses are summarized in Table 3-2. The next chapter explains the experimental design and implementation process.

Table 3-2 Summary of Hypotheses

<table>
<thead>
<tr>
<th>Hypothesis</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hypotheses Regarding Comprehension</td>
<td></td>
</tr>
<tr>
<td>Structural programming task</td>
<td></td>
</tr>
</tbody>
</table>
Table 3-2 Cont.

H1a  In a structural programming task, the use of class diagram will result in a better structural comprehension score than the use of sequence diagram, on average.

H1b  In a structural programming task, the use of class diagram will result in a better structural comprehension score than the use of communication diagram, on average.

H1c  In a structural programming task, the use of communication diagram will result in a better structural comprehension score than the use of sequence diagram, on average.

Behavioral Programming task

H2a  In a behavioral programming task, the use of sequence diagram will result in a better behavioral comprehension score than the use of class diagram, on average.

H2b  In a behavioral programming task, the use of communication diagram will result in a better behavioral comprehension score than the use of class diagram, on average.

H2c  In a behavioral programming task, the use of sequence diagram will result in a better behavioral comprehension score than the use of communication diagram, on average.

Hypotheses Regarding Coding Quality

Structural programming task

H3a  In a structural programming task, the use of class diagram will result in better coding quality than the use of sequence diagram, on average.

H3b  In a structural programming task, the use of class diagram will result in better coding quality than the use of communication diagram, on average.

H3c  In a structural programming task, the use of communication diagram will result in better coding quality than the use of sequence diagram, on average.

Behavioral programming task

H4a  In a behavioral programming task, the use of sequence diagram will result in better coding quality than the use of class diagram, on average.

H4b  In a behavioral programming task, the use of communication diagram will result in better coding quality than the use of class diagram on average.

H4c  In a behavioral programming task, the use of sequence diagram will result in better coding quality than the use of communication diagram, on average.

Hypotheses Regarding Mediation

H5a  In a structural programming task, the structural comprehension mediates the impact of fit on coding quality.

H5b  In a behavioral programming task, the behavioral comprehension mediates the impact of fit on coding quality.
Chapter 4

Research Methodology

In this chapter, the methodology, research setting, design, sample size, measurement, and analysis methods are discussed.

4.1 Methodology and Design

A laboratory experiment was performed to examine the effects of UML diagrams on comprehension and software evolution performance. A laboratory experiment was chosen for the study in order to achieve a high degree of control over extraneous factors likely to affect the relationships of interest.

Two types of task settings were contemplated:

- Structural Task
- Behavioral Task

Three types of UML diagrams were assigned:

- Class Diagram
- Sequence Diagram
- Communication Diagram

Three dependent variables were proposed to be measured:

- Structural Comprehension
- Behavioral Comprehension
- Coding Performance

A three (UML diagrams) by two (task types) fully factorial design was used for the study as shown in Table 4-1.
The experiment was done in the computer lab of a major US University. The computers in the lab were loaded with Java JDK and WordPad. Internet access and subject monitoring was managed through the instructor control center. This prevented access to online Java and UML resources and program solutions. Along with the codes and UML diagrams, minimal documentation help was provided on the Java Class syntax relevant to the problem.

4.2 Experimental Procedure

Each student was assigned randomly to one of the six treatment conditions before the experiment. On the experiment day, the subjects were requested to use the computers randomly assigned to them in the lab. All of the subjects were informed that the experiment consisted of four steps:

1. Forty minutes of UML tutorial
2. Twenty minutes to work on a warm-up task (appendix C–D)
3. Twenty minutes to understand the system with assigned source code and UML diagrams and to answer the comprehension questions (appendix E, G, H, J, L, M)
4. Ninety minutes to complete system modification based on change requirements (appendix F, K).
Individual comprehension level was measured in step 3, and the measurement of programming performance (appendix I, N) was done at the end of the experiment after step 4.

4.3 Experimental Material

A senior faculty member, with expertise in object-orientation, developed the structural and behavioral tasks for the experiment. These two tasks were designed carefully to represent a structural system (appendix E) and a behavioral system (appendix J).

The modification requirements to the structural system were about changing structures. However, it also involved a few process changes (appendix F). The modification requirements to the behavioral system were about changing process and interactions, while some attributes and inheritance structure were required to change as well (appendix K).

The class diagrams, sequence diagrams, and communication diagrams were developed based on the rules and instructions from Booch et al. (1999) and Larman (2004). The faculty expert teaching the system analysis and design course evaluated these diagrams. The diagrams for the structural task are located in appendix G; the diagrams for the behavioral task are located in appendix L.

The subjects in the pilot study were five PhD students and one undergraduate student who were enrolled in information system–related courses. Minor changes were made to improve the experiment procedure and validity of measurement based on the pilot.

4.4 Subjects and Sample Sizes

All subjects participating in the experiment were recruited from an advanced undergraduate Java programming course with approval of their instructors. The subjects’
skill levels and educational background were similar. All subjects were asked to read and sign an informed consent form (appendix A) upon sign-up. As the experiment involved human subjects, prior approval was obtained for the research protocol from the Institutional Review Board (IRB) through the Office of Research Compliance.

Subjects earned class credit for their participation. They were also informed that they would be considered for a lottery prize worth one hundred dollars after completion of the entire experiment. The subjects were also told that the top five scores in the experiment would earn a special prize as well, which served to increase participants’ motivation.

The planned sample size was at least fifteen subjects for each treatment in Table 4-1, totaling a minimum of ninety subjects. Finally, 106 subjects participated in the experiments over two semesters.

4.5 Response Variable Measurements

4.5.1 Comprehension Measures

Comprehension could be measured by free recall (Pennington 1987) and responses to comprehension questions (Corritore et al. 1999). Free recall requires a developer to recall a program. It requires memorization, which may or may not involve understanding, and is used only for small programs. Furthermore, comprehension questions frequently have been employed to assess a maintainer’s internalized representation of the software (Shaft et al. 2006). Hence, the present study measures comprehension by requiring subjects to answer a set of questions while subjects still have access to UML diagrams and the source code.

Comprehension has been separated into structural comprehension and behavioral comprehension, which represent the understanding of different perspectives
of a system. Structural aspects represent the stable skeleton and scaffolding of the system. Structural aspects of a software system encompass the existence and placement of classes, interfaces, relationships, components, and nodes (Booch et al. 1999). Thus structural comprehension measures the subject’s understanding of these items. Questions such as “List all the class/classes associated with account class” are an example of questions used to measure structural comprehension. Behavioral aspects represent the dynamic process of the system. The behavioral aspect of a system encompasses the flow of messages over time, the movement of data across the system, and the role objects play during each process (Booch et al. 1999; Larman 2004). Thus behavioral comprehension should be able to measure: 1) understanding of the logic flow, 2) understanding of the data flow, and 3) understanding of the responsibility of objects (Hadar 2004). Questions are carefully designed to measure these behavioral aspects of a software system, as shown in appendix H and M.

Each subject received both structural and behavioral comprehension questions in a randomized sequence. Subjects answered ten comprehension questions, including five questions measuring structural comprehension and five questions measuring behavioral comprehension.

4.5.2 Coding Performance Measure

Coding performance measured the coding solution of the individual on the programming task. Two common grading schemes were developed in consultation with a faculty expert who teaches an OO programming course. Appendix I shows the grading scheme for structural task, while appendix N shows the grading scheme for behavioral task. In the present study, coding performance measured not only the correctness of the final solution, but also the effectiveness of the solution. For example, as shown in appendix I, item 2 contains two options. Fulfilling any of these options will result in the
correct modification. However, option A is more efficient, because it adopts the inheritance concept and reuses the existing code. Therefore, subjects who accomplished the task using option A earned more points than those who used option B. Similarly, in appendix N, item 5 contains two options as well. Option A intimates an existing similar procedure and makes minor changes to this existing procedure to achieve the new function. Option B rewrites a set of new methods that perform the exact same action as option A. Therefore, subject who used option A gained more credit due to the modification efficiency.

Two doctoral students independently evaluated the software developed by the subjects in the experiment. While grading the pilot test tasks, the grading approaches of the two graders were calibrated by a faculty expert and the experimenter. Separate scoring sheets were created and refined while grading the pilot test tasks. The two doctoral students used the same grading sheets to grade experimental tasks of subjects in the final experiment. The two were requested to reconcile the scores to prevent any errors in grading when large discrepancies were noticed. To reduce any possible sources of bias, the average of the scores of the two raters was used as a measure of software quality.

4.6 Statistical Analysis

MANOVA/ANOVA and general linear regression were the used in data analysis. Two separate one-way MANOVAs were used first to test the overall significance of the model under two different circumstances—improving structural-oriented codes or improving behavioral-oriented codes. After the assumption check and testing the overall significance of the model using MANOVAs, ANOVA models were run to test hypotheses 1–4, which investigate the direct effect of FIT on comprehension and coding performance.
For the structural task, the ANOVA mean model is

\[ Y_{ij} = \mu + \alpha_i + \epsilon_{ij} \quad (i=1, 2, 3; \ j = 1, \ldots, n) \]  

(4.1)

Where \( Y_{ij} \) is the observation of the \( j \)th individual in the \( i \)th group: first group using class diagrams, second group using sequence diagrams, and third group using communication diagrams. The sample size \( (n) \) in each treatment is different, as shown in Table 5-1.

There are sixteen subjects using class diagrams, eighteen subjects using sequence diagrams, and fifteen subjects using communication diagrams. Model (4.1) is translated to the following expression for the general linear model:

\[
\begin{bmatrix}
Y_{1,1} \\
\vdots \\
Y_{1,16} \\
Y_{2,1} \\
\vdots \\
Y_{2,18} \\
Y_{3,1} \\
\vdots \\
Y_{3,15}
\end{bmatrix} =
\begin{bmatrix}
1 & 1 & 0 \\
\vdots & \vdots & \vdots \\
1 & 1 & 0 \\
1 & 0 & 1 \\
\vdots & \vdots & \vdots \\
1 & 0 & 1 \\
1 & -1 & -1 \\
\vdots & \vdots & \vdots \\
1 & -1 & -1
\end{bmatrix}
\begin{bmatrix}
\mu \\
\alpha_i \\
\alpha_2 \\
\epsilon_{1,1} \\
\vdots \\
\epsilon_{1,16} \\
\epsilon_{2,1} \\
\vdots \\
\epsilon_{1,18} \\
\epsilon_{3,1} \\
\vdots \\
\epsilon_{3,15}
\end{bmatrix}
\]  

(4.2)

For the behavioral task, the ANOVA mean model is the same as (4.1). However, the sample size \( (n) \) in each treatment is different, as shown in table 5.1. There are seventeen subjects using class diagrams, fifteen subjects using sequence diagrams, and eighteen subjects using communication diagrams. Using model (4.1) and translating it to the following expression for the general linear model:
Baron and Kenny (1986)'s procedure was adopted to test hypotheses 5a and 5b, which investigate the mediation effect of comprehension. The testing processes for these two hypotheses are same. Taking structural programming tasks as example, three regression models are tested in sequence to find the mediation effect.

The first regression model (M1) is

\[ y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \varepsilon \]  

where \( y \) is the comprehension score, 

and \( x_2 = \begin{cases} 
1 & \text{if } \text{NFIT} \\
0 & \text{if } \text{FIT} 
\end{cases} \). 

The second regression model (M2) is

\[ y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \varepsilon \]  

where \( y \) is coding quality, 

\[ x_1 = \begin{cases} 
1 & \text{if } \text{FIT} \\
0 & \text{if } \text{NFIT} 
\end{cases} \] , and \( x_2 = \begin{cases} 
1 & \text{if } \text{NFIT} \\
0 & \text{if } \text{FIT} 
\end{cases} \). 

The third regression model (M3) is

\[ y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_3 + \varepsilon \]  

where \( y \) is coding quality,
\[ x_1 = \begin{cases} 
1 & \text{if } FIT \\
0 & \text{if } N\text{FIT} 
\end{cases}, \quad x_2 = \begin{cases} 
1 & \text{if } N\text{FIT} \\
0 & \text{if } FIT 
\end{cases} \]

and \( x_3 \) is the comprehension score.

The sample estimates of above models are analyzed in chapter 5, and the interpretations of the results are discussed in chapter 6.
Chapter 5
Research Results

The results of the preliminary analysis and hypothesis testing are presented in this chapter.

5.1 Preliminary Analysis

This section provides the analysis regarding the sample characteristics, assumptions, and preliminary test and analysis methods.

5.1.1 Sample Characteristics

One hundred and six subjects participated in the experiments across two semesters. Three subjects were dropped from the study due to incompleteness of the dependent variable measurement. Four other subjects were considered as outliers because their coding quality, programming, and modeling experiences were abnormally lower than the average. Since the extreme lack of coding and modeling experiences could jeopardize the main effects in the study, those four subjects were dropped from this study. Table 5-1 shows the distribution of subjects across the different treatment conditions.

<table>
<thead>
<tr>
<th>Task Type</th>
<th>Class</th>
<th>Sequence</th>
<th>Communication</th>
</tr>
</thead>
<tbody>
<tr>
<td>Structure Task</td>
<td>16</td>
<td>18</td>
<td>15</td>
</tr>
<tr>
<td>Behavioral Task</td>
<td>17</td>
<td>15</td>
<td>18</td>
</tr>
</tbody>
</table>

Table 5-1 Distribution of Subject across Treatments

The experiment was conducted in two consecutive semesters. Fifty-eight (58.6 percent) valid subjects participated in the spring semester, and forty-one (41.4 percent) valid subjects participated in the following fall semester. Table 5-2 summarizes the results of one-way ANOVA tests for semester on the dependent measures of structural
comprehension, behavioral comprehension, and coding performance. The test result indicates no significant differences in the dependent measures across the two semesters.

Table 5-2 One-way ANOVA Results on Semesters

<table>
<thead>
<tr>
<th>Semester</th>
<th>N</th>
<th>Structural Comprehension</th>
<th>Behavioral Comprehension</th>
<th>Coding Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Mean</td>
<td>SD</td>
<td>Prob &gt;F</td>
</tr>
<tr>
<td>Spring</td>
<td>58</td>
<td>23.07</td>
<td>1.13</td>
<td>0.676</td>
</tr>
<tr>
<td>Fall</td>
<td>41</td>
<td>22.32</td>
<td>1.38</td>
<td>17.18</td>
</tr>
</tbody>
</table>

Several demographic factors were collected during the experiment, including gender, programming experience, and conceptual modeling experience. Table 5-3 summarizes the counts and percentages of total (ninety-nine subjects) for demographic factors by treatment groups.
Table 5-3 Demographics Count Frequency by Treatment ($N = 99$)

<table>
<thead>
<tr>
<th>Factor</th>
<th>Class Diagram</th>
<th></th>
<th></th>
<th>Communication Diagram</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Count</td>
<td>Percentage</td>
<td>Count</td>
<td>Percentage</td>
</tr>
<tr>
<td>Task 1: Structural Task</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Gender</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Female</td>
<td>1</td>
<td>1.01%</td>
<td>4</td>
<td>4.04%</td>
</tr>
<tr>
<td>Male</td>
<td>15</td>
<td>15.15%</td>
<td>14</td>
<td>14.14%</td>
</tr>
<tr>
<td>Programming Experience</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt; 1 year</td>
<td>5</td>
<td>5.05%</td>
<td>8</td>
<td>8.08%</td>
</tr>
<tr>
<td>1–2 years</td>
<td>7</td>
<td>7.07%</td>
<td>6</td>
<td>6.06%</td>
</tr>
<tr>
<td>24 years</td>
<td>3</td>
<td>3.03%</td>
<td>3</td>
<td>3.03%</td>
</tr>
<tr>
<td>&gt; 4 years</td>
<td>1</td>
<td>1.01%</td>
<td>1</td>
<td>1.01%</td>
</tr>
<tr>
<td>Modeling Experience</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt; 1 year</td>
<td>9</td>
<td>9.09%</td>
<td>11</td>
<td>11.11%</td>
</tr>
<tr>
<td>1–2 years</td>
<td>6</td>
<td>6.06%</td>
<td>7</td>
<td>7.07%</td>
</tr>
<tr>
<td>2–4 years</td>
<td>1</td>
<td>1.01%</td>
<td>0</td>
<td>0.00%</td>
</tr>
<tr>
<td>&gt; 4 years</td>
<td>0</td>
<td>0.00%</td>
<td>0</td>
<td>0.00%</td>
</tr>
<tr>
<td>Task 2: Behavioral Task</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Gender</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Female</td>
<td>2</td>
<td>2.02%</td>
<td>2</td>
<td>2.02%</td>
</tr>
<tr>
<td>Male</td>
<td>15</td>
<td>15.15%</td>
<td>13</td>
<td>13.13%</td>
</tr>
<tr>
<td>Programming Experience</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt; 1 year</td>
<td>10</td>
<td>10.10%</td>
<td>2</td>
<td>2.02%</td>
</tr>
<tr>
<td>1–2 years</td>
<td>6</td>
<td>6.06%</td>
<td>6</td>
<td>6.06%</td>
</tr>
<tr>
<td>2–4 years</td>
<td>1</td>
<td>1.01%</td>
<td>7</td>
<td>7.07%</td>
</tr>
<tr>
<td>&gt; 4 years</td>
<td>0</td>
<td>0.00%</td>
<td>0</td>
<td>0.00%</td>
</tr>
<tr>
<td>Modeling Experience</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt; 1 year</td>
<td>14</td>
<td>14.14%</td>
<td>7</td>
<td>7.07%</td>
</tr>
<tr>
<td>1–2 years</td>
<td>3</td>
<td>3.03%</td>
<td>6</td>
<td>6.06%</td>
</tr>
<tr>
<td>2–4 years</td>
<td>0</td>
<td>0.00%</td>
<td>2</td>
<td>2.02%</td>
</tr>
<tr>
<td>&gt; 4 years</td>
<td>0</td>
<td>0.00%</td>
<td>0</td>
<td>0.00%</td>
</tr>
</tbody>
</table>

5.1.2 Assumptions

MANOVA is a robust test. It can stand up to the departures from multivariate normality in terms of Type I error rate (Neter et al. 1996). Statistical power (the power to detect a main or interaction effect) may be reduced when distributions are very plateau-like (Neter et al. 1996). All dependent variables must be distributed normally in each treatment. Four dependent variables were measured in this study. Each subject’s
comprehension of the system was measured in the middle of the experiment before moving on to improve the code. The comprehension consists of two parts: behavior comprehension and structure comprehension. Each subject's coding quality was measured by grading the code after the experiment. Table 5-4 shows the Goodness-of-Fit normality test of four dependent variables in each treatment. The normality assumption could not be rejected for the factor of diagram types (i.e., Class, Sequence, and Communication) at a significance level of $\alpha = 0.05$.

<table>
<thead>
<tr>
<th>DV / Treatment</th>
<th>Class</th>
<th>Sequence</th>
<th>Communication</th>
</tr>
</thead>
<tbody>
<tr>
<td>Structural Task</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total Comprehension</td>
<td>0.3829</td>
<td>0.5365</td>
<td>0.7648</td>
</tr>
<tr>
<td>Structural Comprehension</td>
<td>0.7878</td>
<td>0.0702</td>
<td>0.6550</td>
</tr>
<tr>
<td>Behavioral Comprehension</td>
<td>0.0817</td>
<td>0.7829</td>
<td>0.2962</td>
</tr>
<tr>
<td>Coding Quality</td>
<td>0.5336</td>
<td>0.2666</td>
<td>0.2102</td>
</tr>
<tr>
<td>Behavioral Task</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total Comprehension</td>
<td>0.5448</td>
<td>0.0588</td>
<td>0.7623</td>
</tr>
<tr>
<td>Structural Comprehension</td>
<td>0.1610</td>
<td>0.1759</td>
<td>0.1680</td>
</tr>
<tr>
<td>Behavioral Comprehension</td>
<td>0.2828</td>
<td>0.4490</td>
<td>0.8160</td>
</tr>
<tr>
<td>Coding Quality</td>
<td>0.3983</td>
<td>0.1747</td>
<td>0.1130</td>
</tr>
</tbody>
</table>

Note: $H_0$: the data is from the Normal distribution. Small $p$-values reject $H_0$.

ANOVA analysis requires the variance of dependent variables to be equal across levels of the independent variables (Edwards 1993). MANOVA requires that similar assumptions be met. The univariate requirement of equal variances must hold for each one of the dependent variables. The current MANOVA analysis involved four dependent variables (total comprehension, structure comprehension, behavior comprehension, and code quality) across a three-level factor (class, sequence, and communication diagram). As shown in Table 5-5, The large $p$-value across the three levels on the four dependent variables suggested that the assumption of equal variance could not be rejected, except for the structural comprehension for task 1 ($p$-value = 0.018). The problem of unequal
variance is especially difficult in the presence of unequal sample sizes across treatment groups, and failure of equal variance assumption can lead to serious failure of Type I error control (Edwards 1993). To address this problem, Welch’s test with unequal variance was developed as an alternative to work with unequal variance samples (Welch 1951). Therefore, Welch’s test is used in section 5.1.1.1, in addition to all-pairs comparison procedure. The t-test for unequal variance samples is also used in section 5.1.1.1 to support the comparisons among factor levels.

Table 5-5 Assumption of Equal Variance—Leven p-value

<table>
<thead>
<tr>
<th>Dependent Variable</th>
<th>Leven Test p-value</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Task1: Structural Task</strong></td>
<td></td>
</tr>
<tr>
<td>Total Comprehension</td>
<td>0.0605</td>
</tr>
<tr>
<td>Structural Comprehension</td>
<td>0.0181*</td>
</tr>
<tr>
<td></td>
<td>0.0620a</td>
</tr>
<tr>
<td>Behavioral Comprehension</td>
<td>0.6698</td>
</tr>
<tr>
<td>Coding Quality</td>
<td>0.7464</td>
</tr>
<tr>
<td><strong>Task2: Behavioral Task</strong></td>
<td></td>
</tr>
<tr>
<td>Total Comprehension</td>
<td>0.6829</td>
</tr>
<tr>
<td>Structural Comprehension</td>
<td>0.8722</td>
</tr>
<tr>
<td>Behavioral Comprehension</td>
<td>0.0860</td>
</tr>
<tr>
<td>Coding Quality</td>
<td>0.5336</td>
</tr>
</tbody>
</table>

Note: H0= three groups have the same variance. 
*: Small p-values reject H0. 
a: Results of O’Brien test is non-significant.

5.1.3 Reliability of Dependent Measures

Comprehension and coding quality were measured at the individual level. The internal consistency of the measurements was tested by Pearson correlation, which is an indicator of dependency among measured variables (Cohen et al. 2002).
For comprehension measurement, each subject was required to answer ten questions regarding the code in the middle of the experiment. Half of the questions captured structural information, while the other half captured process information in the code. All the answers to those questions are objective and straightforward. Two doctoral teaching assistants graded the answers. The internal consistency of the comprehension scores was checked using Pearson correlation with a value of 0.962.

For coding quality, each subject was required to improve existing codes in two hours. Two doctoral teaching assistants graded the improved codes following a grading scheme. The internal consistency of the coding quality scores was checked using Pearson correlation with a value of 0.808. Pearson correlation values over 0.7 are indications of a strong positive relationship between the two measurements (Dowdy et al. 1983). Thus it is adequate to assume measurement consistency and reliability in this study.

5.1.4 Test of Significance—Main Effects

Two one-way MANOVAs were conducted separately to examine the main effect of diagram type on multiple dependent variables under two different circumstances: improving structure-oriented codes or improving behavior-oriented codes. Therefore, the fit between diagram type and task type existed, by nature, in each task. For structural task (task 1), the fit between diagrams and task type existed for class diagram. When conducting behavioral task (task 2), the fit between diagrams and task type existed only for sequence diagram. Therefore, although the diagrams were labeled as class, sequence, and communication in the study, they represented the FIT as well. For task 1, class diagram represented structural fit (SFIT), sequence diagram represented non-fit (NFIT), and communication diagram represented partial fit (PFIT). Similarly, for task 2, class diagram represented NFIT, sequence diagram represented behavioral fit (BFIT),
and communication diagram represented PFIT. However, the fit could not be tested as an interaction effect in this study because the two tasks are incomparable. In other words, the question “Would the behavioral fit facilitate coding better than structural fit?” will not be answered in this study. Only the differences among FIT, NFIT, and PFIT within the same task were tested in this study. To better elaborate the relationship, the label was changed from types of diagrams to types of fit. Table 5-6 shows the new labels mapping to types of diagrams. The labels were used interchangeably in the following discussion.

<table>
<thead>
<tr>
<th>Task 1: Structural Task</th>
<th>Task 2: Behavioral Task</th>
</tr>
</thead>
<tbody>
<tr>
<td>New Label</td>
<td>Old Label</td>
</tr>
<tr>
<td>SFIT</td>
<td>Class Diagram</td>
</tr>
<tr>
<td>NFIT</td>
<td>Sequence Diagram</td>
</tr>
<tr>
<td>PFIT</td>
<td>Communication Diagram</td>
</tr>
</tbody>
</table>

For both tasks, one-way MANOVA analysis was conducted separately on four dependent variables (Total Comprehension, Structure Comprehension, Behavior Comprehension, and Coding Quality). The overall F-test (overall four dependent variables) was found to be significant with low Wilks’s Lambda p-value for both tasks. The Lambda is a measure of the percent of variance in the dependent variables that is not explained by differences in the level of independent variable. It means, for structural task, only 19.4 percent of variances in dependent variables were not explained by level of fit, associated with F of 13.676, which was significant at p-value <0.000. For behavioral task, over 40 percent of variances in dependent variables were not explained by level of fit. The unexplained variances were larger than those of structural task but still associated with F of 6.344, which were significant at p-value <0.000.
Table 5-7 MANCOVA Results for Dependent Variables

<table>
<thead>
<tr>
<th></th>
<th>Value</th>
<th>F-Value</th>
<th>Between Group</th>
<th>Within Group</th>
<th>P-value</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Task 1—FIT vs. Dependent Variables</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Wilks’s Lambda</td>
<td>0.194</td>
<td>13.676</td>
<td>8</td>
<td>88</td>
<td>0.000*</td>
</tr>
<tr>
<td>Pillai’s Trace</td>
<td>0.999</td>
<td>10.967</td>
<td>8</td>
<td>88</td>
<td>0.000*</td>
</tr>
<tr>
<td>Hotelling-Lawley</td>
<td>3.171</td>
<td>16.818</td>
<td>8</td>
<td>60</td>
<td>0.000*</td>
</tr>
<tr>
<td>Roy’s Max Root</td>
<td>2.819</td>
<td>31.003</td>
<td>4</td>
<td>44</td>
<td>0.000*</td>
</tr>
<tr>
<td><strong>Task 2—FIT vs. Dependent Variables</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Wilks’s Lambda</td>
<td>0.402</td>
<td>6.344</td>
<td>8</td>
<td>88</td>
<td>0.000*</td>
</tr>
<tr>
<td>Pillai’s Trace</td>
<td>0.629</td>
<td>5.169</td>
<td>8</td>
<td>88</td>
<td>0.000*</td>
</tr>
<tr>
<td>Hotelling-Lawley</td>
<td>1.407</td>
<td>7.637</td>
<td>8</td>
<td>60</td>
<td>0.000*</td>
</tr>
<tr>
<td>Roy’s Max Root</td>
<td>1.348</td>
<td>15.165</td>
<td>4</td>
<td>45</td>
<td>0.000*</td>
</tr>
</tbody>
</table>

* Significant at p-value = 0.05

Table 5-7 revealed a significant main effect of fit type on the four dependent variables. Further one-way ANOVAs were conducted to examine the dependent variables differences among factor levels (FIT, NFIT, and PFIT).

The one-way ANOVAs results are summarized in Table 5-8. It revealed that the main effects of types of fit were different under various circumstances. When performing structural task (Task 1), the subjects’ structure comprehension, behavior comprehension, and coding quality varied according to the type of fit. However, there were no significant differences in total comprehension among the subjects across types of fit (p-value = 0.2164). The possible reason was that the total comprehension value was a sum of structure comprehension value and behavior comprehension value. Table 5-8 also revealed that the subjects using class diagrams (SFIT) gained high structure comprehension scores and low behavior comprehension scores, while the subjects using sequence diagrams (NFIT) gained low structure comprehension scores and high
behavior comprehension scores. When adding structure comprehension and behavioral comprehension, the differences canceled out for total comprehension.

Table 5-8 One-Way ANOVA Results on Dependent Variables—Factor Level

<table>
<thead>
<tr>
<th>Task1: Structural Task</th>
<th>FIT</th>
<th>NFIT</th>
<th>PFIT</th>
<th>F-Value</th>
<th>p-value</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Mean</td>
<td>SD</td>
<td>Mean</td>
<td>SD</td>
<td>Mean</td>
</tr>
<tr>
<td>TComp</td>
<td>47.59</td>
<td>2.43</td>
<td>42.86</td>
<td>2.50</td>
<td>42.00</td>
</tr>
<tr>
<td>SComp</td>
<td>32.97</td>
<td>1.32</td>
<td>24.77</td>
<td>1.36</td>
<td>19.02</td>
</tr>
<tr>
<td>BComp</td>
<td>14.62</td>
<td>1.97</td>
<td>18.01</td>
<td>2.04</td>
<td>22.97</td>
</tr>
<tr>
<td>CodingQuality</td>
<td>58.94</td>
<td>3.05</td>
<td>57.97</td>
<td>3.15</td>
<td>32.03</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Task2: Behavioral Task</th>
<th>FIT</th>
<th>NFIT</th>
<th>PFIT</th>
<th>F-Value</th>
<th>p-value</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Mean</td>
<td>SD</td>
<td>Mean</td>
<td>SD</td>
<td>Mean</td>
</tr>
<tr>
<td>TComp</td>
<td>30.62</td>
<td>2.42</td>
<td>33.60</td>
<td>2.36</td>
<td>43.32</td>
</tr>
<tr>
<td>SComp</td>
<td>22.49</td>
<td>2.13</td>
<td>17.95</td>
<td>2.07</td>
<td>20.45</td>
</tr>
<tr>
<td>BComp</td>
<td>8.10</td>
<td>1.43</td>
<td>15.65</td>
<td>1.38</td>
<td>22.88</td>
</tr>
<tr>
<td>CodingQuality</td>
<td>42.50</td>
<td>3.30</td>
<td>53.19</td>
<td>3.21</td>
<td>66.13</td>
</tr>
</tbody>
</table>

When performing a behavioral task (Task 2), the subjects’ total comprehension, behavior comprehension, and coding quality varied according to the type of fit. However, there were no significant differences of structure comprehension among the subjects using different type of diagrams (p-value = 0.319). The possible reason was that the second task was simple in terms of structure. When the structure of the code was simple, it was easy to capture the structural information with or without the help of proper tools.

Generally, the preliminary results were found to be consistent with our theorization. Further examination of which diagram was significantly better will be conducted in the hypothesis testing section 5.2

5.1.5 Test of Significance—Correlations

In order to perform regression analysis, the type of fit (FIT, NFIT, PFIT) was coded using two dummy variables. The variable labeled “FIT” was coded as either 1 (FIT) or 0 (the others). The variable labeled “NFIT” was coded as either 1 (NFIT) or 0 (the
others). In other words, the two dummy variables represented the three types of fit as shown in Table 5-9.

<table>
<thead>
<tr>
<th>Dummy Variables</th>
<th>FIT</th>
<th>NFIT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fit</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Non-Fit</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Partial Fit</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

The correlations among independent and dependent variables were examined under two different circumstances—improving structure-oriented code and improving behavior-oriented code. The results are summarized in Table 5-10.
Table 5-10 Significant of Correlation

<table>
<thead>
<tr>
<th>Task 1: Structural Task</th>
<th>Mean</th>
<th>Std Dev</th>
<th>(1)</th>
<th>(2)</th>
<th>(3)</th>
<th>(4)</th>
<th>(5)</th>
<th>(6)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1) Total Comprehension</td>
<td>44.087</td>
<td>9.822</td>
<td>1.000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(2) Structure Comprehension</td>
<td>25.337</td>
<td>7.827</td>
<td>0.558</td>
<td>1.000</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(3) Behavior Comprehension</td>
<td>18.730</td>
<td>8.505</td>
<td>0.647</td>
<td>-0.271</td>
<td>1.000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(4) Coding Quality</td>
<td>48.755</td>
<td>17.568</td>
<td>0.491</td>
<td>0.641</td>
<td>-0.020</td>
<td>1.000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(5) FIT</td>
<td>0.327</td>
<td>0.474</td>
<td>0.251</td>
<td>0.686</td>
<td>-0.340</td>
<td>0.408</td>
<td>1.000</td>
<td></td>
</tr>
<tr>
<td>(6) NFIT</td>
<td>0.367</td>
<td>0.487</td>
<td>-0.164</td>
<td>-0.621</td>
<td>0.384</td>
<td>-0.733</td>
<td>-0.531</td>
<td>1.000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Task 2: Behavioral Task</th>
<th>Mean</th>
<th>Std Dev</th>
<th>(1)</th>
<th>(2)</th>
<th>(3)</th>
<th>(4)</th>
<th>(5)</th>
<th>(6)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1) Total Comprehension</td>
<td>35.505</td>
<td>11.142</td>
<td>1.000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(2) Structure Comprehension</td>
<td>20.245</td>
<td>8.798</td>
<td>0.678</td>
<td>1.000</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(3) Behavior Comprehension</td>
<td>15.250</td>
<td>8.290</td>
<td>0.625</td>
<td>-0.150</td>
<td>1.000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(4) Coding Quality</td>
<td>53.440</td>
<td>16.380</td>
<td>0.565</td>
<td>0.189</td>
<td>0.560</td>
<td>1.000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(5) NFIT</td>
<td>0.340</td>
<td>0.479</td>
<td>-0.318</td>
<td>0.185</td>
<td>-0.625</td>
<td>-0.484</td>
<td>1.000</td>
<td></td>
</tr>
<tr>
<td>(6) FIT</td>
<td>0.300</td>
<td>0.463</td>
<td>0.464</td>
<td>0.015</td>
<td>0.608</td>
<td>0.513</td>
<td>-0.470</td>
<td>1.000</td>
</tr>
</tbody>
</table>

* Significant at level of 0.05
For task 1, both total comprehension (0.491, 0.000*) and structural comprehension (0.641, 0.000*) were related to coding quality positively and significantly at a level of 0.05. The test showed a negative but no significant relationship between behavioral comprehension and coding quality (-0.020, 0.890). It indicated that, for structural task, understanding structural aspects of the system was more important than understanding behavioral aspects of the system. The type of fit also influenced the structural comprehension score, behavioral comprehension score, and coding quality significantly. FIT was correlated with structural comprehension score (0.686, 0.000*) and coding quality (0.408, 0.004*) positively and significantly. However, both FIT and NFIT were not found to be correlated with total comprehension significantly.

For task 2, both total comprehension (0.565, 0.000*) and behavioral comprehension (0.560, 0.000*) were related to coding quality positively and significantly. There was no significant evidence to show the impact of structural comprehension on coding quality (0.189, 0.189). It revealed that for behavioral task, understanding behavioral aspects of the system was more important than understanding the structural aspects of the system.

5.2 Hypothesis Testing

Hypothesis testing consisted of two parts. First, one-way MANOVA and ANOVA models were used to examine the main effects of fit on each dependent variable (Structural Comprehension, Behavioral Comprehension, and Code Quality). As discussed in Table 5-7, for both tasks, the overall F-test was found to be significant. Both of these one-way MANOVA revealed a significant main effect of FIT. To further test the hypothesized family of inferences, Tukey-Kramer multiple inference procedure was conducted. Tukey-Kramer procedures were used because they are superior to other multiple inference procedures when all pairwise comparisons were of interest (Neter et al.
1996). Welch’s test is also used for the comparison of structural comprehension since the unequal variance is likely to be found across fit levels for structural comprehension.

Second, multiple regression analysis was conducted to examine the hypothesized mediator effect of comprehension on the relationship between type of FIT and coding quality. Three regression equations were built in sequence in order to test for statistically significant mediator effects (Baron et al. 1986).

5.2.1 Hypotheses Concerning Comprehension

In this section, multiple comparisons were conducted to test hypothesized differences for the dependent variables of structural comprehension and behavioral comprehension. For the structural task, FIT has been proposed to influence structural comprehension score \((H1a, 1b, 1c)\), while for the behavioral task, FIT has been proposed to influence the behavioral comprehension score \((H1d, 1e, 1f)\). The Tukey-Kramer multiple inference procedure was conducted to examine the differences among FIT levels.

5.2.1.1 Task1: Structural Programming Task \((H1a, 1b, 1c)\)

Structural comprehension was estimated as the average of the scores reported by the two graders who independently graded the comprehension questions. While conducting the structural programming task, FIT existed for subjects using class diagrams. The subject using sequence diagrams should gain the limited benefit of diagrams due to the lack of fit between task and diagram type. The subject using the communication diagrams fell onto the partial fit factor level. Significant main effects for FIT were obtained for structural comprehension score, \(F = 28.89\), and \(p\)-value \(<.0001^*\) adjusted for one-tailed test. Significant Tukey-Kramer pairwise comparisons are obtained between FIT, NFIT, and PFIT groups, which are summarized in Table 5-11. The \(p\)-values in Table 5-11 were adjusted for one-tailed test since all comparisons are directional.
Table 5-11 Tukey-Kramer Comparison—Structural Comprehension Scores (α = 0.05)

<table>
<thead>
<tr>
<th>Level</th>
<th>- Level Difference</th>
<th>Std Err Dif</th>
<th>Lower CL</th>
<th>Upper CL</th>
<th>p-Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIT</td>
<td>NFIT</td>
<td>13.94097</td>
<td>1.838</td>
<td>9.489</td>
<td>.0001*</td>
</tr>
<tr>
<td>FIT</td>
<td>PFIT</td>
<td>8.26875</td>
<td>1.923</td>
<td>3.612</td>
<td>0.0002*</td>
</tr>
<tr>
<td>PFIT</td>
<td>NFIT</td>
<td>5.67222</td>
<td>1.871</td>
<td>1.142</td>
<td>0.0054*</td>
</tr>
</tbody>
</table>

As found in section 5.1.2, unequal variance is likely to be found across fit levels for structural comprehension scores when performing structural task. Welch’s test, which allows unequal variance, is also done in addition to F-test. The p-value is the probability of obtaining an F-value larger than the one calculated, if in reality the means are equal across all levels. Observed significance probabilities of 0.05 or less are considered evidence of unequal means across the levels. Therefore, the reported small p-value (<.0001*) associated with large F-ratio (43.9255) is a strong indication of unequal means across three fit levels. The t-test for unequal variance is also done for each hypothesis in section 5.1.1.1.1–section 5.1.1.1.3 to ensure that the unequal variance issue has been taken care of.

5.2.1.1 Hypothesis H1a

*H1a: In a structural programming task, the use of class diagram would result in a better structural comprehension score than the use of sequence diagram, on average.*

Table 5-12 Means for Structural Comprehension Score—H1a

<table>
<thead>
<tr>
<th>Factor Level</th>
<th>Mean</th>
<th>Std Error</th>
<th>Difference</th>
<th>Prob &gt; t</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIT (Class Diagram)</td>
<td>32.969</td>
<td>1.3386</td>
<td>13.941</td>
<td>&lt;.0001*</td>
</tr>
<tr>
<td>NFIT (Sequence Diagram)</td>
<td>24.700</td>
<td>1.382</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

In a structural programming task, the group using class diagram was included in FIT factor level, while the group using sequence diagram was included in NFIT factor...
level. Table 5-12 shows the relevant means and comparison results. The structural comprehension score of FIT group was found to be higher than that of NFIT group by 13.94 on average, associated with significant $p$-value <0.0001*. The t-test for unequal variance also reported a small $p$-value <0.0001*.

_Hypothesis 1a_ was therefore fully supported. Consequently, the use of class diagram would result in a better structural comprehension score than the use of sequence diagram, on average, when the subjects worked on a structural programming task. Figure 5-1 Marginal Means of Structural Comprehension Score

Figure 5-1 shows the plot of the marginal means of structural comprehension score.

![Figure 5-1 Marginal Means of Structural Comprehension Score—Structural Task](image)

**5.2.1.1.2 Hypothesis H1b**

_H1b: In a structural programming task, the use of class diagram would result in better structural comprehension score than the use of communication diagram, on average._

In a structural programming task, the group using class diagram was included in FIT factor level, while the group using communication diagram was included in Partial Fit (PFIT) factor level. Table 5-13 shows the relevant means and comparison results. As proposed, the structural comprehension score of FIT group was found to be higher than
that of PFIT group by 8.268 on average, associated with significant \( p \)-value = 0.0002*.

The t-test for unequal variance also reported a small and significant \( p \)-value < 0.001*. Hypothesis 1b was therefore fully supported. Consequently, the use of class diagram would result in a better structural comprehension score than the use of communication diagram, on average, when the subjects worked on a structural programming task. Figure 5-1 showed the plot of the marginal means of structural comprehension score.

Table 5-13 Means for Structural Comprehension Score—H1b

<table>
<thead>
<tr>
<th>Factor Level</th>
<th>Mean</th>
<th>Std Error</th>
<th>Difference</th>
<th>Prob &gt; t</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIT (Class Diagram)</td>
<td>32.969</td>
<td>1.3376</td>
<td>8.26875</td>
<td>0.0002*</td>
</tr>
<tr>
<td>PFIT (Communication Diagram)</td>
<td>19.028</td>
<td>1.2611</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

5.2.1.1.3 Hypothesis H1c

\[ H1c: \text{In a structural programming task, the use of communication diagram would result in a better structural comprehension score than the use of sequence diagram, on average.} \]

Table 5-14 Means for Structural Comprehension Score—H1c

<table>
<thead>
<tr>
<th>Factor Level</th>
<th>Mean</th>
<th>Std Error</th>
<th>Difference</th>
<th>Prob &gt; t</th>
</tr>
</thead>
<tbody>
<tr>
<td>PFIT (Communication Diagram)</td>
<td>24.700</td>
<td>1.382</td>
<td>5.672</td>
<td>0.0054*</td>
</tr>
<tr>
<td>NFIT (Sequence Diagram)</td>
<td>19.028</td>
<td>1.261</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 5-14 shows the relevant means and comparison results. As proposed, the structural comprehension score of NFIT group was found to be higher than that of PFIT group by 8.268 on average, associated with significant \( p \)-value = 0.0054*. The t-test for unequal variance also reported a significant \( p \)-value = 0.0490*. Hypothesis 1c was therefore fully supported. Consequently, the use of communication diagram would result
in better structural comprehension score than the use of sequence diagram, on average, when the subjects worked on a structural programming task. Figure 5-1 showed the plot of the marginal means of structural comprehension score.

5.2.1.2 Task 2: Behavioral Programming task (H2a, 2b, 2c)

Behavioral comprehension was estimated as the average score provided by the two raters who independently graded the comprehension questions. While conducting a behavioral programming task, FIT existed for subjects using sequence diagram. The subject using class diagrams should gain the least benefit of diagrams, due to the lack of fit between task and diagram type. The subject using the communication diagrams was included in the partial fit factor level. Significant main effects for FIT level were obtained for the behavioral comprehension score, $F = 25.22$, and $p < .0001^*$. Significant Tukey-Kramer pairwise comparisons were obtained between the FIT, NFIT and PFIT groups, which are summarized in Table 5-15 with $p$-value adjusted for the one-tailed test.

Table 5-15 Tukey-Kramer Comparison of Structural Comprehension Scores ($\alpha = 0.05$)

<table>
<thead>
<tr>
<th>Level</th>
<th>Level Difference</th>
<th>Std Err Dif</th>
<th>Lower CL</th>
<th>Upper CL</th>
<th>Prob &gt; t</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIT</td>
<td>NFIT</td>
<td>14.772</td>
<td>2.0824</td>
<td>9.732</td>
<td>19.812</td>
</tr>
<tr>
<td>PFIT</td>
<td>NFIT</td>
<td>7.543</td>
<td>1.9881</td>
<td>2.731</td>
<td>12.354</td>
</tr>
<tr>
<td>FIT</td>
<td>PFIT</td>
<td>7.229</td>
<td>2.0551</td>
<td>2.255</td>
<td>12.203</td>
</tr>
</tbody>
</table>

5.2.1.2.1 Hypothesis H2a

$H2a$: In a behavioral programming task, the use of sequence diagram would result in a better behavioral comprehension score than the use of class diagram, on average.

Table 5-16 Means for Structural Comprehension Score—$H2a$

<table>
<thead>
<tr>
<th>Factor Level</th>
<th>Mean</th>
<th>Std Error</th>
<th>Difference</th>
<th>Prob &gt; t</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIT (Sequence Diagram)</td>
<td>22.875</td>
<td>1.518</td>
<td>14.772</td>
<td>&lt;.0001*</td>
</tr>
</tbody>
</table>
Table 5-16 Cont.

<table>
<thead>
<tr>
<th>Factor Level</th>
<th>Mean</th>
<th>Std Error</th>
<th>Difference</th>
<th>Prob &gt; t</th>
</tr>
</thead>
<tbody>
<tr>
<td>NFIT (Class Diagram)</td>
<td>8.103</td>
<td>1.426</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

In a behavioral programming task, the group using sequence diagram was FIT factor level, while the group using class diagram was included in the NFIT factor level.

Table 5-16 shows the relevant means and comparison results. The structural comprehension score of the FIT group was found to be higher than that of the NFIT group by 14.77, on average, associated with significant $p$-value $<$0.0001*, adjusted for the one-tailed test. Hypothesis 2a is consequently fully supported. Therefore, the use of sequence diagram would result in a better behavioral comprehension score than the use of class diagram, on average, when the subjects worked on a behavioral programming task. Figure 5-2 shows the plot of the marginal means of structural comprehension score.

![Figure 5-2 Marginal Means of Structural Comprehension Score—Behavioral Task](image)

5.2.1.2.2 Hypothesis H2b

$H2b$: *In a behavioral programming task, the use of sequence diagram will result in a better behavioral comprehension score than the use of communication diagram, on average.*
Table 5-17 Means for Structural Comprehension Score—$H2b$

<table>
<thead>
<tr>
<th>Factor Level</th>
<th>Mean</th>
<th>Std Error</th>
<th>Difference</th>
<th>Prob &gt; t</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIT (Sequence Diagram)</td>
<td>22.88</td>
<td>1.52</td>
<td>7.23</td>
<td>0.0014*</td>
</tr>
<tr>
<td>PFIT (Communication Diagram)</td>
<td>15.65</td>
<td>1.39</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

In a behavioral programming task, the group using sequence diagram had a FIT while the group using communication diagram had a partial fit (PFIT). Table 5-17 shows the relevant means and comparison results. As proposed, the structural comprehension score of the FIT group was found to be higher than that of the PFIT group by .23, on average, associated with significant $p$-value = 0.0014*, adjusted for the one-tailed test. 

Hypothesis 2b was therefore fully supported. Consequently, the use of sequence diagram would result in better behavioral comprehension score than the use of communication diagram, on average, when the subjects worked on a structural programming task. Figure 5-2 shows the plot of the marginal means of structural comprehension score.

5.2.1.2.3 Hypothesis $H2c$

$H2c$: *In a behavioral programming task, the use of communication diagram will result in better behavioral comprehension score than the use of class diagram, on average.*

Table 5-18 Means for Structural Comprehension Score—$H2c$

<table>
<thead>
<tr>
<th>Factor Level</th>
<th>Mean</th>
<th>Std Error</th>
<th>Difference</th>
<th>Prob &gt; t</th>
</tr>
</thead>
<tbody>
<tr>
<td>PFIT (Communication Diagram)</td>
<td>15.65</td>
<td>1.39</td>
<td>7.54</td>
<td>0.0006*</td>
</tr>
<tr>
<td>NFIT (Class Diagram)</td>
<td>8.10</td>
<td>1.40</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 5-18 shows the relevant means and comparison results. As proposed, the structural comprehension score of NFIT group was found to be higher than that of PFIT group by 8.268 on average, associated with significant $p$-value = 0.0006*, adjusted for
the one-tailed test. *Hypothesis 2c* was therefore fully supported. Consequently, the use of communication diagram would result in a better structural comprehension score than the use of sequence diagram, on average, when the subjects worked on a structural programming task. Figure 5-2 showed the plot of the marginal means of structural comprehension score.

5.2.2 Hypotheses Concerning Code Quality

In this section, ANOVA models were conducted to test hypothesized comparison on the dependent variable of code quality. Code quality was estimated by grading subjects' codes according to a detailed grading scheme. The code quality was calculated as the average score assigned by the two independent graders.

For both the structural and behavioral programming tasks, fit level had been proposed to influence coding quality (*H3a–3c, H4a–4c*). Tukey-Kramer multiple inference procedure was conducted to examine the coding quality differences among FIT levels.

5.2.2.1 Task1: Structural Programming Task

Given the significance of the overall Wilks's Lambda test, the main effect for FIT was obtained by Tukey-Kramer multiple inference procedure. Similar to the analysis in section 5.1.1.1, while conducting a structural programming task, FIT existed for subjects using class diagram. The subject using sequence diagrams should have no benefit due to a lack of fit between the task and the diagram type. The subject using communication diagrams fell onto the partial fit factor level. Significant main effects for FIT were obtained for coding quality, *F* = 22.61, and *p* < .0001*. Tukey-Kramer pairwise comparisons were obtained between FIT, NFIT, and PFIT groups, which are summarized in Table 5-19 with a justified one-tailed *p*-value.
Table 5-19 Tukey-Kramer Comparison Report of Average Coding Quality at $\alpha = 0.05$

<table>
<thead>
<tr>
<th>Level</th>
<th>- Level Difference</th>
<th>Std Err Dif</th>
<th>Lower CL</th>
<th>Upper CL</th>
<th>Prob &gt; t</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIT</td>
<td>NFIT</td>
<td>26.910</td>
<td>4.484</td>
<td>16.050</td>
<td>37.770</td>
</tr>
<tr>
<td>PFIT</td>
<td>NFIT</td>
<td>24.939</td>
<td>4.563</td>
<td>13.889</td>
<td>35.989</td>
</tr>
<tr>
<td>FIT</td>
<td>PFIT</td>
<td>1.9708</td>
<td>4.690</td>
<td>-9.389</td>
<td>13.330</td>
</tr>
</tbody>
</table>

5.2.2.1.1 Hypothesis H3a

H3a: In a structural programming task, the use of class diagram will result in better coding quality than the use of sequence diagram, on average.

Table 5-20 Means for Coding Quality—H3a

<table>
<thead>
<tr>
<th>Factor Level</th>
<th>Mean</th>
<th>Std Error</th>
<th>Difference</th>
<th>Prob &gt; t</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIT (Class Diagram)</td>
<td>58.938</td>
<td>3.263</td>
<td>26.910</td>
<td>&lt;.0001*</td>
</tr>
<tr>
<td>NFIT (Sequence Diagram)</td>
<td>32.028</td>
<td>3.076</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 5-20 shows the relevant means and comparison results. The coding quality of FIT group was found to be higher than that of NFIT group by 26.91, on average, associated with significant $p$-value <0.0001*, adjusted for the one-tailed test. Hypothesis 3a was therefore fully supported. Therefore, the use of class diagram would result in better coding quality than the use of sequence diagram, on average, when the subjects worked on a structural programming task. Figure 5-3 showed the plot of the marginal means of coding quality.
Figure 5-3 Marginal Means of Coding Quality—Structural Task

5.2.2.1.2 Hypothesis H3b

H3b: In a structural programming task, the use of class diagram will result in better coding quality than the use of communication diagram, on average.

Table 5-21 Means for Coding Quality—H3b

<table>
<thead>
<tr>
<th>Factor Level</th>
<th>Mean</th>
<th>Std Error</th>
<th>Difference</th>
<th>Prob &gt; t</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIT (Class Diagram)</td>
<td>58.938</td>
<td>3.263</td>
<td>1.971</td>
<td>0.454</td>
</tr>
<tr>
<td>PFIT (Communication Diagram)</td>
<td>56.970</td>
<td>3.370</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 5-21 shows the relevant means and comparison results. No evidence was found to reject the hypothesis that the coding qualities of FIT group and PFIT group were different. The differences were found to be 1.971, on average, with an adjusted one-tailed p-value = 0.4538. Hypothesis 3b was therefore not supported. Figure 5-3 showed the plot of the marginal means of coding quality.

5.2.2.1.3 Hypothesis H3c

H3c: In a structural programming task, the use of communication diagram will result in better coding quality than the use of sequence diagram, on average.
Table 5-22 Means for Coding Quality—H3c

<table>
<thead>
<tr>
<th>Factor Level</th>
<th>Mean</th>
<th>Std Error</th>
<th>Difference</th>
<th>Prob &gt; t</th>
</tr>
</thead>
<tbody>
<tr>
<td>PFIT (Communication Diagram)</td>
<td>56.97</td>
<td>3.370</td>
<td>24.892</td>
<td>&lt;.0001*</td>
</tr>
<tr>
<td>NFIT (Sequence Diagram)</td>
<td>32.028</td>
<td>3.076</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 5-22 shows the relevant means and comparison results. The coding quality of PFIT group was found to be higher than that of NFIT group by 24.93889 on average, associated with significant p-value <0.0001*, adjusted for the one-tailed test. *Hypothesis 3c was therefore fully supported. Consequently, the use of communication diagram would result in better coding quality than the use of sequence diagram, on average, when the subjects worked on the structural programming task. Figure 5-3 shows the plot of the marginal means of coding quality.

5.2.2.2 Task 2: Behavioral Programming task

While conducting a behavioral programming task, FIT existed for subjects using sequence diagram. The subject using class diagrams should gain the least benefit of diagrams due to the lack of fit between task and diagram type. The subject using the communication diagrams fell to the partial fit factor level. Significant main effects for FIT level were obtained for coding quality, $F = 12.04$, $p < .0001^*$. Significant Tukey-Kramer pairwise comparisons were obtained between the FIT, NFIT, and PFIT groups, which are summarized in Table 5-23. Figure 5-4 showed the plot of the marginal means of coding quality. Tukey-Kramer pairwise comparisons were obtained between the FIT, NFIT, and PFIT groups, which are summarized in Table 5-23 with a justified one-tailed $p$-value.
Table 5-23 Tukey-Kramer Comparison Report of Coding Quality at $\alpha = 0.05$

<table>
<thead>
<tr>
<th>Level</th>
<th>Level Difference</th>
<th>Std Err Dif</th>
<th>Lower CL</th>
<th>Upper CL</th>
<th>Porb &gt; t</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIT</td>
<td>FIT</td>
<td>23.633</td>
<td>4.818</td>
<td>11.973</td>
<td>35.293</td>
</tr>
<tr>
<td>FIT</td>
<td>PFIT</td>
<td>12.939</td>
<td>4.755</td>
<td>1.432</td>
<td>24.446</td>
</tr>
<tr>
<td>PFIT</td>
<td>NFIT</td>
<td>10.694</td>
<td>4.600</td>
<td>-0.437</td>
<td>21.826</td>
</tr>
</tbody>
</table>

Figure 5-4 Marginal Means of Coding Quality—Behavioral Task

5.2.2.2.1 Hypothesis H4a

$H4a$: In a behavioral programming task, the use of sequence diagram will result in better coding quality than the use of class diagram, on average.

Table 5-24 Means for Coding Quality—H4a

<table>
<thead>
<tr>
<th>Factor Level</th>
<th>Mean</th>
<th>Std Error</th>
<th>Difference</th>
<th>Prob &gt; t</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIT (Sequence Diagram)</td>
<td>66.133</td>
<td>3.512</td>
<td>23.633</td>
<td>&lt;.0001*</td>
</tr>
<tr>
<td>NFIT (Class Diagram)</td>
<td>42.500</td>
<td>3.299</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 5-24 shows the relevant means and comparison results. The coding quality of the FIT group was found to be higher than that of the NFIT group by 23.633, on average, associated with a significant one-tailed $p$-value <0.0001*. Hypothesis 2d was therefore fully supported. Consequently, the use of sequence diagram would result in better coding quality than the use of class diagram, on average, when the subjects worked on a behavioral programming task.
5.2.2.2.2 Hypothesis H4b

H4b: In a behavioral programming task, the use of sequence diagram will result in better coding quality than the use of communication diagram, on average.

Table 5-25 Means for Coding Quality—H4b

<table>
<thead>
<tr>
<th>Factor Level</th>
<th>Mean</th>
<th>Std Error</th>
<th>Difference</th>
<th>Prob &gt; t</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIT (Sequence Diagram)</td>
<td>66.133</td>
<td>3.512</td>
<td>12.939</td>
<td>0.0121*</td>
</tr>
<tr>
<td>PFIT (Communication Diagram)</td>
<td>42.500</td>
<td>3.299</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 5-25 shows the relevant means and comparison results. As proposed, the coding quality of the FIT group was found to be higher than that of the PFIT group by 12.939, on average, associated with a significant one-tailed p-value = 0.0121*.

Hypothesis 4b was therefore fully supported. Consequently, the use of sequence diagram would result in better coding quality than the use of communication diagram, on average, when the subjects worked on a behavioral programming task.

5.2.2.2.3 Hypothesis H4c

H4c: In a behavioral programming task, the use of communication diagram will result in better coding quality than the use of class diagram, on average.

Table 5-26 Means for Coding Quality—H4c

<table>
<thead>
<tr>
<th>Factor Level</th>
<th>Mean</th>
<th>Std Error</th>
<th>Difference</th>
<th>Prob &gt; t</th>
</tr>
</thead>
<tbody>
<tr>
<td>PFIT (Communication Diagram)</td>
<td>66.133</td>
<td>3.512</td>
<td>10.695</td>
<td>0.0620</td>
</tr>
<tr>
<td>NFIT (Class Diagram)</td>
<td>42.500</td>
<td>3.299</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 5-26 shows the relevant means and comparison results. The coding quality of the PFIT group was better than that of the NFIT group, as proposed. However, the differences were not significant with p-value = 0.0620 at the significant level of 0.05.
Therefore, there was not enough evidence to fully support $H4c$ at $\alpha = .05$, although the hypothesis is supported at $\alpha = .10$.

5.2.3 Hypotheses Concerning Mediation Effect of Comprehension

In this section, the mediation effect of comprehension between FIT and coding quality was examined separately for structural programming tasks and behavioral programming tasks. The mediator effects were tested with multiple regression analysis.

Mediator effects should be tested if there is a significant direct association between the input variable and the output variable. Otherwise there is no relationship to mediate. In this study, the input variable was a manipulated category FIT type with three levels (FIT, NFIT, and PFIT). In order to perform multiple regression analysis, this category variable was coded by two dummy variables. The variable labeled “FIT” was coded as either 1 (FIT) or 0 (the others). The variable labeled as “NFIT” was coded as either 1 (NFIT) or 0 (others). In other words, the two dummy variables represented the three types of fit, as shown in Table 5-9 in section 5.1.7. The output variable in this study was coding quality. The single order relationships among these variables were confirmed by statistically significant Pearson correlations in the expected direction (Table 5-11 in section 5.1.7). As predicted, the FIT/NFIT variables and coding quality were correlated for both structural programming task ($r=0.408 / -0.733$) and behavioral programming task ($r=0.513/-0.484$).

Significant direct associations were also required between mediators and output variable. For the structural programming task, the structural comprehension score was proposed as the mediator, and it was found to associate with coding quality ($r=0.491$). For the behavioral programming task, the behavioral comprehension score was proposed as the mediator, and it was found to correlate with coding quality ($r=0.560$). The full Pearson correlations report can be found in Table 5-10. For both programming tasks, the
total comprehension score was proposed as the mediator as well. The total comprehension scores were associated with a significant positive impact of coding quality for structural programming task \((r=0.491)\) and behavioral programming task \((r=0.565)\).

According to Baron and Kenny (1986), once the correlations are found, the next step is to use the three regression models (4.5), (4.6), and (4.7) to test the mediator effect. If the mediator effect is present, two conditions must be met in M3. First, the mediator must be a significant predictor of the outcome variable. Second, if the full mediator effect exists, the effect of input variable on outcome-dependent variable should be non-significant. Baron and Kenny argue that the strongest evidence for mediation is when this partial effect of X on Y is reduced to non-significance, but partial mediation effect also exists, as long as the effect of input variable on outcome variable controlling for mediator is less than the direct effect of input variable on output variable without controlling for mediator. The three steps were followed and executed for structural programming task and behavioral programming task separately in the following two sections. Figure 5.8 shows the regression models to be tested among input variable, mediator, and the outcome variable.

![Figure 5-5 Mediator Test Models](image)

5.2.3.1 Hypothesis 5a

\(H5a: \text{In a structural programming task, the structural comprehension mediates the impact of fit on coding quality.}\)
Table 5-27 shows the overall F-values, the R^2's, and parameter estimates with t-value and p-value for M1, M2, and M3, respectively. M1 was significant, associated with F=28.90, p<0.0001. M2 regressed coding quality on fit levels, associated with F=22.61 and p<0.0001.

Since the fit levels were a three-level category variable and coded as two separated dummy variables, one significant estimate out of the two dummy variables would indicate that the fit levels overall were a significant predictor of coding quality. As shown in Table 5-27, in the second model M2, the effects on coding quality of FIT and PFIT are not significantly different from each other, but NFIT has a significantly different effect on coding quality. M3 regressed dependent variable coding quality with fit levels controlling for structural comprehension. The model was significant with F=21.95 and p<0.0001. These predictors explained 59.40 percent of the variance in coding quality. These patterns of results conformed to the criteria established by Baron and Kenny (1986) for partial mediation.

In addition, M3 met two requirements for a partial mediator effect. First, the hypothesized mediator Structural Comprehension was a significant predictor of outcome variable Coding Quality (β-hat=1.077 and p-value = 0.002*) that explained 43.52 percent of the variance. Second, the fit levels were still a significant predictor, as NFIT β-hat = -18.831 and p-value = 0.000. However, the percentage of variance explained by fit levels (FIT and NFIT) was reduced from 49.57 percent in M2 to 15.88 percent in M3 controlling for the mediator. Thus the reduced direct association between the fit levels and coding quality when structural comprehension was controlled in M3 partially supported the hypothesis H5a that structural comprehension mediates the relationship between fit levels and coding quality partially.
Table 5-27 Summary of Regression Analysis—H5a

<table>
<thead>
<tr>
<th>Model</th>
<th>F-Value</th>
<th>R Square</th>
<th>Estimates $\hat{\beta}$</th>
<th>t-Value</th>
<th>Prob &gt; t</th>
<th>Prob &lt; - t</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>DV: Structural Comprehension</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M1</td>
<td>28.90*</td>
<td>0.5568</td>
<td>8.269</td>
<td>4.30</td>
<td>&lt;0.001*</td>
<td>0.002*</td>
</tr>
<tr>
<td>FIT</td>
<td></td>
<td></td>
<td>-5.672</td>
<td>-3.03</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NFIT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>DV: Code Quality</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M2</td>
<td>22.61*</td>
<td>0.4957</td>
<td>0.1621</td>
<td>1.970</td>
<td>0.42</td>
<td>0.3281</td>
</tr>
<tr>
<td>FIT</td>
<td></td>
<td></td>
<td>0.3275</td>
<td>-24.94</td>
<td>-5.47</td>
<td>0.000*</td>
</tr>
<tr>
<td>NFIT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M3</td>
<td>21.95*</td>
<td>0.5940</td>
<td>0.4352</td>
<td>1.077</td>
<td>0.326</td>
<td>0.001*</td>
</tr>
<tr>
<td>Structural Comprehension</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FIT</td>
<td></td>
<td></td>
<td>0.0032</td>
<td>-6.933</td>
<td>0.5038</td>
<td>0.0878</td>
</tr>
<tr>
<td>NFIT</td>
<td></td>
<td></td>
<td>0.1556</td>
<td>-18.831</td>
<td>4.534</td>
<td>0.000*</td>
</tr>
</tbody>
</table>

5.2.3.2 Hypothesis 5b

*H5b: In a behavioral programming task, the behavioral comprehension mediates the impact of fit on coding quality.*

A similar procedure was performed to evaluate H5b. Table 5-28 shows the overall F-values, the $R^2$s, and parameter estimates with t-value and $p$-value for M1, M2, and M3, respectively. M1 was significant, associated with F=25.22 and $p < 0.001$. M2 regressed coding quality on fit levels associated with F=12.04 and $p < 0.001$. M3 regressed coding quality on fit levels controlling for behavioral comprehension. The model was significant with F=23.32 and $p<0.0001$.

In addition, M3 met two requirements for a fully mediated effect. First, the hypothesized mediator Structural Comprehension was a significant predictor of outcome variable Coding Quality ($\hat{\beta}$-hat=0.584 and $p$-value = 0.042*), which explained 31.4 percent of the variance. Second, the fit levels become insignificant predictors, as FIT $\hat{\beta}$-
hat = 8.716 with p-value = 0.055, and NFIT β-hat = -6.289 with p-value = 0.114. Thus

H5b fully supports that behavioral comprehension fully mediates the relationship between fit levels and coding quality in behavioral programming tasks.

Table 5-28 Summary of Regression Analysis—H3b

<table>
<thead>
<tr>
<th>Model</th>
<th>F-Value</th>
<th>R Square</th>
<th>Estimates β</th>
<th>t-Value</th>
<th>Prob &gt; t</th>
<th>Prob &lt; - t</th>
</tr>
</thead>
<tbody>
<tr>
<td>DV: Behavioral Comprehension</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M1</td>
<td>25.22*</td>
<td>0.518</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FIT</td>
<td>0.370</td>
<td>7.229</td>
<td>3.52</td>
<td>0.001*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NFIT</td>
<td>0.148</td>
<td>-7.543</td>
<td>-3.79</td>
<td>0.000*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DV: Code Quality</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M2</td>
<td>12.04*</td>
<td>0.339</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FIT</td>
<td>0.263</td>
<td>12.939</td>
<td>2.72</td>
<td>0.009*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NFIT</td>
<td>0.076</td>
<td>-10.694</td>
<td>-2.33</td>
<td>0.024*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>M3</td>
<td>23.32*</td>
<td>0.380</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Behavioral Comprehension</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FIT</td>
<td>0.314</td>
<td>0.584</td>
<td>1.77</td>
<td>0.042*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NFIT</td>
<td>0.075</td>
<td>8.716</td>
<td>1.67</td>
<td>0.055</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

5.2.4 Summary of Hypothesis Test

The results of hypothesis testing are summarized in Table 5-29. Three hypotheses in all were not supported. The detailed discussions of these exceptions are presented in Chapter 6.
Table 5-29 Results of Hypothesis Testing

<table>
<thead>
<tr>
<th>Hypotheses Regarding Comprehension</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>H1a</strong> In a structural programming task, the use of class diagram will result in a better structural comprehension score than the use of sequence diagram, on average.</td>
<td>Supported</td>
</tr>
<tr>
<td><strong>H1b</strong> In a structural programming task, the use of class diagram will result in a better structural comprehension score than the use of communication diagram, on average.</td>
<td>Supported</td>
</tr>
<tr>
<td><strong>H1c</strong> In a structural programming task, the use of communication diagram will result in a better structural comprehension score than the use of sequence diagram, on average.</td>
<td>Supported</td>
</tr>
<tr>
<td><strong>H2a</strong> In a behavioral programming task, the use of sequence diagram will result in a better behavioral comprehension score than the use of class diagram, on average.</td>
<td>Supported</td>
</tr>
<tr>
<td><strong>H2b</strong> In a behavioral programming task, the use of sequence diagram will result in a better behavioral comprehension score than the use of communication diagram, on average.</td>
<td>Supported</td>
</tr>
<tr>
<td><strong>H2c</strong> In a behavioral programming task, the use of communication diagram will result in a better behavioral comprehension score than the use of class diagram, on average.</td>
<td>Supported</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Hypotheses Regarding Coding Quality</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>H3a</strong> In a structural programming task, the use of class diagram will result in better coding quality than the use of sequence diagram, on average.</td>
<td>Supported</td>
</tr>
<tr>
<td><strong>H3b</strong> In a structural programming task, the use of class diagram will result in better coding quality than the use of communication diagram, on average.</td>
<td>Not Supported</td>
</tr>
<tr>
<td><strong>H3c</strong> In a structural programming task, the use of communication diagram will result in better coding quality than the use of sequence diagram, on average.</td>
<td>Supported</td>
</tr>
<tr>
<td><strong>H4a</strong> In a behavioral programming task, the use of sequence diagram will result in better coding quality than the use of class diagram, on average.</td>
<td>Supported</td>
</tr>
<tr>
<td><strong>H4b</strong> In a behavioral programming task, the use of sequence diagram will result in better coding quality than the use of communication diagram, on average.</td>
<td>Supported</td>
</tr>
<tr>
<td><strong>H4c</strong> In a behavioral programming task, the use of communication diagram will result in better coding quality than the use of class diagram, on average.</td>
<td>Not Supported</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Hypotheses Regarding Mediation</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>H5a</strong> In a structural programming task, the structural comprehension mediates the impact of fit on coding quality.</td>
<td>Partially Supported</td>
</tr>
<tr>
<td><strong>H5b</strong> In a behavioral programming task, the behavioral comprehension mediates the impact of fit on coding quality.</td>
<td>Supported</td>
</tr>
</tbody>
</table>
Chapter 6
Summary, Limitations, and Future Research

The fundamental question of the present study is to investigate whether or not representation matters in the contingence of task type. UML diagrams, an external representation of reality, are poor and incomplete in practice. Though UML diagrams facilitate design, specification, and testing during the software development life cycle, they are expensive and are difficult artifacts to maintain under the typical time pressure in the software industry (Briand 2003). Developers and maintainers think that UML diagrams are useful, but there is little agreement on what is most useful under different situations. Normally, a system could generate UML diagrams automatically, based on source code, to minimize the maintenance effort for the developer. However, when the maintainers try to pick up the job later, they are overwhelmed by the size of UML diagrams and normally turn directly to the code (Grossman et al. 2005).

Maintainers may turn back to certain UML diagrams after reading the code and find them easier to comprehend or useful for locating the codes to be modified. The challenge is to understand with minimum comprehension effort when to use an UML diagram and which UML diagram type is required to find the best coding solution. Our findings suggest that developers do not need all types of UML diagrams. Developers would reduce documentation effort by focusing on certain types of UML diagrams for different types of task when maintaining software.

6.1 Summary of Research Findings

This section discusses the experimental results on the effect of FIT on comprehension and coding performance. The mediation effect of comprehension is also presented.
6.1.1 Regarding the Comprehension

When subjects worked on the structural-oriented task, the average structural comprehension was 32.97 for class diagram users, 19.03 for sequence diagram users, and 24.70 for communication diagram users. The advantage of class diagram over the other two diagrams was significant at the level of 0.05. The advantage of communication diagrams over sequence diagrams on structural comprehension was also significant at the level of 0.05. The results strongly indicate that if the goal is to achieve useful comprehension of the system then maintainers should choose class diagrams or communication diagrams rather than sequence diagrams. Class diagram is the best tool to achieve a comprehensive understanding of the structural aspects of the system.

When subjects worked on the behavioral-oriented task, the average behavioral comprehension were 8.10 for class diagram users, 22.88 for sequence diagram users and 15.65 for communication diagram users. The subjects using class diagram achieved a behavioral comprehension level significantly lower than the other subjects using sequence or communication diagrams. The difference between sequence diagram users and communication diagram users was also significant at the level of 0.05. The findings are consistent with theoretical predictions. It indicated that, to achieve better behavioral comprehension, the usages of sequence diagram or communication diagram is preferred when performing a structural-oriented programming task. Class diagram does not significantly contribute to behavioral comprehension.

The overall comprehension was also tested, even though it is not part of the hypotheses. When performing structural task, the overall comprehension on average were 47.59 for class diagrams, 37.03 for sequence diagrams and 47.74 for communication diagrams. Not surprisingly, the overall comprehension for the subjects using class diagram and communication diagrams were not significantly different from
each other. The main reason is that the total comprehension value is a sum of structural comprehension value and behavioral comprehension value. Subjects using class diagrams (FIT) gained high structure comprehension scores and low behavior comprehension scores, whereas those using communication diagrams (PFIT) gained relatively low structure comprehension scores but high behavior comprehension scores. When adding together structure comprehension and behavioral comprehension, the sum cancels out the differences of total comprehension scores across types of fit. It further validated the reason to study structural and behavioral comprehension separately.

Previous similar studies focused on overall comprehension, which resulted in insignificant results (Glezer et al. 2005; Nugroho 2009).

6.1.2 Regarding Coding Quality

The direct effect of FIT on coding quality was tested through hypotheses H3a–H3c and H4a–H4c. When subjects worked on a structural-oriented task, the average coding quality were 58.94 for class diagram users, 32.03 for sequence diagram users, and 56.97 for communication diagram users. The subjects assigned with class diagrams and communication diagrams performed significantly better than those assigned with sequence diagrams at the level of 0.05. This result strongly indicates that sequence diagrams are not particularly helpful and can be skipped for the structural-oriented task. However, class diagrams and communication diagrams are similar in the way they facilitate a structural coding task. However, H3b was not supported, as there was no significant coding performance differences between the subjects using class diagrams and communication diagrams at the level of 0.05.

When the subjects worked on behavioral-oriented task, the average coding quality were 42.50 for class diagrams, 66.13 for sequence diagrams, and 53.19 for communication diagrams. The use of the sequence diagram is significantly better in
achieving coding quality than the use of the other two diagrams. The use of communication diagram is better in achieving coding quality than the use of class diagram; however, this advantage is not significant as proposed in $H4c$.

In sum, the study failed to prove hypotheses $H3b$ and $H4c$. These two hypotheses are a comparison between class diagrams and communication diagrams under different task environments. No salient differences were found between these two diagrams under either task environment—structural or behavioral. There are two reasons to explain the unexpected result.

First, communication diagrams contain structural components. Maintainers could obtain a similar amount of structural information directly from the class diagram, or from multiple communication diagrams. For the structural task, even the design contained a complicated structure, but the overall system was relatively simple. Therefore, the effort spent on reading one class diagram was not much different from the efforts spent on exploring multiple communication diagrams. A more complicated system may be able to reveal the salient advantages of using class diagram for structural tasks in the future.

Second, communication diagrams are interchangeable with sequence diagrams; however, the representations of system behaviors in communication diagrams are not as direct as in sequence diagrams. Sequence diagrams display the time sequence of objects participating in each function horizontally, and the sequence of messages shows how the system behaves. This representation style matches all the sequences encoded in the source code, and it helps maintainers locate the behavior that needs modification in source code. Communication diagrams contain the same behavioral information as sequence diagrams, in that the salient difference of behavioral comprehension was found between class and communication diagrams ($H2c$ is supported). However, the communication diagrams represent the behavioral information
in the form of class diagrams (e.g., using hierarchical structure rather than horizontal sequential messages); therefore maintainers found it easy to gain behavioral comprehension but difficult to locate the place of the behavior needed for the modification in the source code. This is why no significant differences of coding performance were identified between subjects using class diagrams and sequence diagrams when performing a behavioral task.

6.1.3 Regarding the Mediation Effect of Comprehension

Linear regressions were conducted to test the mediation effect of comprehension. When the subject performs a structural task, the structural comprehension positively and partially mediates the relationship between fit and coding performance (H5a). Partial mediation implies that there is not only a significant relationship between structural comprehension and coding performance but also a direct relationship between fit levels and coding performance. In other words, when performing a structural task, the maintainer could improve the coding performance by enhancing the understanding of structural aspects of the system, or by choosing the correct tool to fit the task. With the correct choice of tool, maintainers could maintain minimum effort in understanding the system but pay more attention to coding. Class diagrams and communication diagrams are two options for maintainers to gain accurate and efficient structural understanding.

When the subject performs a behavioral-orientated task, behavioral comprehension was found to positively and fully mediate the relationship between fit levels and coding performance (H5b). The full mediation implies that behavioral comprehension alone would be enough to explain all the variances in coding performance. The impact of fit levels on coding performance became weak and insignificant with the introduction of behavioral comprehension. In other words, when
performing a behavioral task, in order to improve the coding performance, the maintainers should choose the right tool to increase the behavioral comprehension first, before starting to modify the source code.

The findings indicated that for a structural task, using structural diagrams to locate the place to make changes directly is more efficient than using these diagrams to produce a solution through enhancing comprehension. Maintainers should be able to change the code effectively and efficiently with a structural external representation tool, like class diagram, without spending too much effort on understanding the system. For a behavioral task, in contrast, gaining comprehension is the required step between choosing the right tool and improving coding performance.

6.1.4 Summary of Findings

In summary, when performing structural programming tasks, subjects using any diagrams emphasizing the structural aspect of the system (e.g., class diagram and communication diagram) would achieve both higher structural comprehension and better coding quality than those using the diagrams emphasizing behavioral aspects of the system (e.g., sequence diagram). Structural comprehension was found to partially mediate the relationship between structural fit and coding quality. Class diagrams and communication diagrams have a similar impact on structural comprehension and coding quality because of the simplicity of the structural task. If the structural task was more complicated, it could be easier to reveal the salient difference between class diagram and communication diagram. Therefore, the best way to improve coding quality for a structural-oriented programming task is through choosing the right tool to improve coding performance with minimum comprehension efforts.

When performing behavioral-oriented programming tasks, subjects using class diagrams achieved behavioral comprehension significantly lower than those using
sequence or communication diagrams. However, behavioral comprehension was significantly and fully mediated between behavioral fit and coding performance. Therefore, for behavioral task, improved coding performance through gaining comprehension is more efficient than using diagrams to locate the place to make changes directly.

6.2 Significance of Findings

The present study is developed based on a few excellent research studies (see for example, Agarwal et al. 1999; Khatri et al. 2006a; Shaft et al. 2006). However, there are some limitations to these existing studies. First, most of the existing studies focused either on the effect of conceptual models on comprehension task (Agarwal et al. 1999; Shaft et al. 2006) or the direct effect of conceptual models on maintenance tasks (Khatri et al. 2006a). Actually, the relationship between comprehension and coding performance is much more complicated than those findings. The present study also makes substantial improvement by examining structural and behavioral comprehension separately. It makes sure that the two effects would not cancel each other, which led to an insignificant finding in previous studies (Glezer et al. 2005; Nugroho 2009). The present study suggests that the two types of comprehension are the mediators between task fit and coding performance. The mediation effect is more significant when performing a behavioral task. When performing a structural task, it is less important to gain structural comprehension than to choose the right modeling tool to perform the modification task directly.

Second, the OO programming paradigm was neglected in all existing studies (Agarwal et al. 1999; Khatri et al. 2006a; Shaft et al. 2006). Accordingly, the conceptual modeling tools under investigation are out-of-date in these studies. Instead of ER and DF models, more OO models should be studied. UML, one of the most popular tools conceptualizing a domain for OO programming, has not been widely studied in the
contingency of task type. The present study fills these gaps in research by investigating
the relationship among UML diagrams, comprehension task, and object-oriented coding
task.

The positive findings in the present study also confirmed the importance of the
distributed cognitive theory, cognitive fit theory, and task-technology fit theory in the
information systems research field. The task-technology fit is a straightforward theory that
argues that a tool can have positive impact on performance only if the tool fits the task
(Goodhue 2006). The distributed cognitive theory and the cognitive fit theory can be used
to explain why and how the task-technology fit has an impact on performance. These two
theories open the black box between the task-technology fit and performance. They both
assert that subjects can achieve better performance through gaining more understanding
of the domain, and some representations can help subjects gaining the domain
knowledge. Zhang and Norman (1994)'s distributed cognition theory argues that external
representation matters. Representing the same problem in different forms will result in
different perceptions of the task. Cognitive fit theory proposes that the correspondence
between internal representation of the task and external information presentation format
leads to superior task performance for individual users (Shaft et al. 2006). These theories
can be used to explain more interesting but unclear questions in the field of information
systems. More artifacts should be explored. For example, since Agile development
resists all documented external representations, what is the best conceptual
representation for an Agile development project? How does one best utilize the tool to
cope with Agile development characteristics? The process should be opened up more by
asking questions such as whether the usage patterns of conceptual representations are
different given different task types.
6.3 Implications for Practitioners

Conceptual models make complex problems easy because external rules can be checked by perceptual inspection directly, without overloading working memory. From this perspective, researchers are right that conceptual models, like UML, are necessary to developers’ understanding of reality through searching useful information among diagrams. However, in practice, it requires too many conceptual models to understand a system comprehensively. External representations cannot improve understanding unlimitedly. There are some points to which external representations are too much for an individual to inspect directly. It will overload the working memory and deteriorate the cognition process. Therefore, from this perspective, the practitioners are right in that too many diagram documents serve only as hurdles to them, rather than facilitators. Overloaded diagrams stop maintainers from enjoying the benefits that come along with diagrammatical conceptual models. The present study settled this conflict by investigating the impact of different types of diagrammatic conceptual models on software modification performance under various circumstances. In addition, choosing the right diagrams in certain situations may not improve performance. The black box between diagram usage and performance was opened and examined. The study answers the following three questions which are meaningful to practitioners:

1. Does the choice of diagrammatical conceptual models affect software modification performance?
2. Is the choice contingent upon the nature of the programming task?
3. Does the programmer’s comprehension level mediate the association between the choice of conceptual models and software maintenance performance?
The findings suggest that not all diagrams are necessary at the same time. The choice of the diagram depends on the nature of the programming task (structural or behavioral) and the final output (comprehension or coding quality). Two basic practice rules were raised based on theoretical findings:

If the maintainers work with a structurally complicated system, they should use structural models, like UML class diagram, to locate the places in source code requiring modification. Structural comprehension is important, but a minimum effort of understanding the structural aspect of the system is good enough.

If the maintainers work with a behaviorally complicated system, enhanced behavioral comprehension of the system is a required antecedent to improved coding performance in this circumstance. Maintainers should use behavioral models, such as sequence or communication diagrams, to gain behavioral comprehension of the system first. Coding performance is dependent on the level of positive behavioral comprehension.

This experimental study simulated two extreme conditions: a structural system and a behavioral system. However, in practice, no single system is purely structural or purely behavioral. Therefore, it is possible to combine these two rules together in practice. If the maintainers feel it necessary to gain detailed understanding of the system, they should adopt behavioral diagrams. If the maintainers want to fix or add structural components to the system, they should use class diagrams directly, without trying to comprehend first. In this way, maintainers could choose the most useful diagrams, instead of becoming overwhelmed by too many diagrams.

While the present study emphasizes the importance of UML diagrams, it also brings up another issue—the quality of UML diagrams. Existing studies suggest that modelers would benefit from models only when the models manifest a good and simple
representation of reality (Burton-Jones et al. 2008). There are many research trends to discuss how to improve the model quality. Factors such as level of decomposition (Burton-Jones et al. 2008; Formica et al. 2004), level of details (Nugroho 2009), and notations (Gerard 2005; Kabeli et al. 2005) will affect the quality of an UML diagram. In order to gain the benefits of UML diagrams, practitioners must develop good-quality UML diagrams first following the guidelines provided by the research paper.

In addition, the significant findings in the present study indicate that not all diagrams are useful and only selected ones should be used to achieve good quality with minimum comprehension effort. The findings are also a hint to teachers in the field of education. With limited classroom hours, teachers should educate students, who are the future system developers and maintainers, to focus on those conceptual models that are most effective and useful in practice. The importance of class diagrams, sequence diagrams, and communication diagrams is found to vary depending upon the type of task in the present study; however, more types of external representations should be explored to help students keep up with what is happening in industry.

6.4 Limitations

First, the tasks designed for the experiment utilized a familiar domain-banking system. It raised these questions: Would the results be different if the application domain is unfamiliar? Would fit play a different role on comprehension and coding quality?

Second, the structural task in this experiment is a structurally complicated task, but it was not as complicated and comparable as the behavioral task. It is essential to design comparable tasks in the future study in order to compare performance across tasks.

Third, in the present study, the four students with over four years of coding experience achieved significantly higher structural comprehension scores and coding
performance than those with less coding experience. It indicates that expertise level is also an important factor leading to different comprehension and coding levels. Given small numbers of expertise in the present study, it is impossible to find any significant results. Future study should include more expertise in the study in order to compare the differences between novices and experts. It is an exciting topic to investigate. Does the effect of external representation on comprehension and coding performance vary between novices and experts?

Finally, software developers working in the industry would have made the experiment results more convincing and generalizable. Use of students as subjects is a limitation of this study. The use of college students as subjects might raise issues of generalizability. It might be argued that students cannot be considered as good subject choices since they do not possess the experience or expertise to match with software developers who have been working in the industry. However, some factors might actually be considered in favor of student subjects. Software professionals in the field setting may come with different demographic and educational backgrounds, which would prevent researchers from investigating factors of interest (Joshi et al. 2007). For example, software professionals who have been working using one type of methodology might develop habitual routines specific to that methodology and might try to build code from that mind-set in the case they are randomly assigned to work, following a different methodology. It might lead to bias in data collected. However, it is still valuable to gain a more comprehensive and generalizable understanding of the effectiveness of external representations in different field settings. By doing this, researchers would gain a better position to make practical recommendations to practitioners regarding the choice of conceptual models.
Appendix A

Enrollment Flyer and Informed Consent
Free UML Seminar

Dr. Radha Mahapatra, Dr. Sridhar Nerur and Ms. Lulu Zhang of the University of Texas at Arlington are conducting a study to understand the role of diagrammatic representation in object-oriented development. We are currently recruiting volunteers to participate in this study.

Participants will receive a free seminar on UML topics including class diagram, sequence diagram and communication diagram.

Participants will also work individually to solve object-oriented programming problems and will fill out a questionnaire.

Eligibility

Eligible participants should have taken Java 2 class or had equivalent Java programming experience.

UML experience is a plus, but is not required. The UML seminar will provide sufficient information.

Contact us

Ms. Lulu Zhang
Phone: 817-272-3584
Email: lulu.zhang@mavs.uta.edu
Office: COBA 533

All eligible participants will be entered into a prize draw for an 8GB iPod Shuffle with five runner-up prizes of a $10 gift card.

Enrollment

BY FILLING IN AND RETURNING THIS FORM YOU CONSENT THE FOLLOWING:

Consent to attend the seminar and work individually to fill out a questionnaire and solve the programming problems.

Name: ______________
NetID: ______________
Email: ______________
1st Choice:
Date: 26th April, Tuesday
Time: 3:30pm – 6:30pm

2nd Choice:
Day: 30th April 2011, Sat
Time: 9am – 12:00pm

Signature: ______________
Appendix B

Debriefing
Debriefing

In this study, we were interested in examining the effectiveness of conceptual models in programming. We were looking to see if different graphical models facilitated system comprehension and programming performance variously under different circumstance (structural or behavioral system). UML diagrams were used as an example of the conceptual modeling tool.

To examine this question we randomly selected which individuals are allotted to class diagram, sequence diagram, or communication diagram condition and structural task condition or behavioral task condition. Your understanding of the system and programming performance will be used to judge the effect of various factors on the effectiveness of UML diagrams.

We will be conducting this study with more students in the fall 2011. It is vitally important to the success of this experiment that you keep the information you have learned here in confidence. Please do not tell anyone about the content of this experiment. We are confident that we can trust you. Thank you.

Please sign below to indicate that you understand this debriefing and that you promise to keep what you have learned in confidence. Again, thank you very much for your participation.

Signature ______________________

Date ______________________
Appendix C

Warm-up Task Descriptions and Requirements
Vehicle Registration System Description

The vehicle registration office is in the process of upgrading its vehicle registration application. The current system allows agents to register vehicles for customers.

Currently, the system allows agents to perform following actions:

Action 1–3: Create Vehicle Owner/Vehicle/Registration
   The system allows agents to create vehicle owner, vehicle, and registration.
   Each customer may have zero or more vehicle registrations.

Action 4: Display Owner
   The system displays owner information

Action 5: Display Vehicle
   The system displays vehicle information.

Action 6: Display Registration
   The system displays registration information with associated owner and vehicle information.

Enhancement Tasks:
The current system does not allow retrieval of registration through vehicle. This capability will be added with the following steps:

1. Add a string attribute to the vehicle class - RegistrationNumber
2. When a vehicle object is created, the initial RegistrationNumber is 0.
3. Once the registration the vehicle object has been completed, the system will update the RegistrationNumber in the vehicle object.
Appendix D

Warm-up Task—UML Diagrams
Class Diagram

Vehicle

- id: String
- make: String
- model: String
- year: int

- Vehicle()
  - Vehicle(String id, String make, String model, int year)
  - setVehicleID(): String
  - setMake(): String
  - setModel(): String
  - setYear(): int

- toString(): String

Registration

- registrationNumber: String
- registrationDate: Date

- Registration()
  - Registration(String registrationNumber, Date registrationDate)

Owner

- ownerID: String
- ownerName: String
- ownerAddress: String

- Owner()
  - Owner(String ownerID, String ownerName, String ownerAddress)

Application

- createOwner(): String
  - createRegistration(): String

Create Owner

Create Vehicle
**Display Registration**

1. DisplayRegistration()
2. aRegistration = getRegistration()
3. writeRegistration(aRegistration)

**Communication Diagrams**

### Create Owner

1. createOwner()

2. <<Create>> (id, name, address)

3. put(id, anOwner)

anOwner : Owner

### Create Vehicle

1. createVehicle()

2. <<Create>> (vehicleID, make, model, year)

3. put(id, aVehicle)

aVehicle : Vehicle
Create Registration

1. createRegistration()  
2. amOwner = getOwner()  
3. aVehicle = getVehicle()  
4. <<Create>> (amOwner, aVehicle)  
5. putRegNumber, aRegistration

Display Owner

1. DisplayOwner()  
2. amOwner = getOwner()  
3. writeOwner(amOwner)

Display Vehicle

1. DisplayVehicle()  
2. aVehicle = getVehicle()  
3. writeVehicle(aVehicle)

Display Registration

1. DisplayRegistration()  
2. aRegistration = getRegistration()  
3. writeRegistration(aRegistration)
Appendix E

Task 1: Structural Task—Description
System Description

ABC Bank is in the process of upgrading its banking application. Currently, the system allows bank agents to perform following actions:

Action 1: Create New Customer
The system allows bank agents to create two types of customers: personal and business. The bank agent assigns each new customer with a customer ID.

Action 2: Create New Account
The system allows bank agents to create new accounts. One or more accounts may be created for each customer. The system assigns each new account with an account number.

Action 3: Deposit Transaction
Deposit transaction can be performed on an account. The system assigns each new deposit transaction with a transaction number.

Action 4: Withdrawal Transaction
Withdrawal transaction can be performed on an account. The system assigns each new withdrawal transaction with a transaction number.

Action 5: Display Customer Information
The system displays customer information using customer ID.

Action 6: Display Account Information
The system displays account information with associated customer information using account number.
Appendix F

Task 1: Structural Task—Modification Requirements
Enhancement Task Description

In order to be competitive in the marketplace, ABC Bank is trying to add some new capabilities and features to their application. The new requirements are listed below:

1. When creating an account, the bank agent will be asked to open a savings or checking account. In addition,
   a. Checking account must maintain a minimum balance of $250.00. In other words, when an account is created, the minimum balance is $250.00. Likewise, when customer withdraws money, the system should ensure that the balance in checking account does not fall below $250.00.
   b. Savings accounts cannot have a negative balance. The system should not allow customers to withdraw more than what they have in their savings accounts.
   c. When displaying account details, the system will show account type.
   d. When displaying transaction details after each withdrawal or deposit transaction, the system will show account type.

2. The system will allow a new action. Action 7: Display Transaction Information. In other words, the system will retrieve and display transaction information using transaction number.
Appendix G

Task 1: Structural Task—UML Diagrams
Class Diagram

Visual Paradigm for UML Community Edition (not for commercial use)

Sequence Diagrams

Create New Personal Customer

Visual Paradigm for UML Community Edition (not for commercial use)
Deposit Transaction

1: deposit()

2: anAccount = findAccount()

3: <<Create>> anAccount, "Deposit", amount

4: execute()

5: deposit(amount)

6: setBalance(finalBalance)

Withdrawal Transaction

1: withdraw

2: anAccount = findAccount()

3: <<Create>> anAccount, "Withdraw", amount

4: execute()

5: withdraw(amount)

6: setBalance(finalBalance)
Display Account

1. displayAccount()

2. anAccount = findAccount()

3. writeAccount(anAccount)

Display Customer

1. displayCustomer()

2. aCustomer = findCustomer()

3. writeCustomer()

Communication Diagrams

Create New Personal Customer

1. CreateCustomer()

2. <<Create>> (Name, ID, Address)

3. saveCustomer(aCustomer)

Create New Business Customer

1. CreateCustomer()

2. <<Create>> (Name, ID, Address, OrgName, OrgID, OrgAddress)

3. saveCustomer(aCustomer)

Create New Account

1. createNewAccount()

2. aCustomer = findCustomer()

3. <<Create>> (aCustomer, InitialBalance)

4. saveAccount(anAccount)

Save new customer ID

Hiltable

anAccount : Account
Deposit Transaction

Withdrawal Transaction

Display Account

Display Customer
Appendix H

Task 1: Structural Task—Comprehension Questions and Rubrics
<table>
<thead>
<tr>
<th>Question</th>
<th>Answer</th>
<th>Perfect Points</th>
</tr>
</thead>
</table>
| 1. List the sequence of methods involved in creating a new account object. | menu()  
run()  
createAccount  
findCustomer()  
<<create>>anAccount = new Account(...)  
addAccount(...) | 1  
1  
2  
2  
2 |
|                                                                         | Other methods:  
accountsTable.put(...)  
setCustomer(...)  
getResponse() | 0  
0  
0 |
|                                                                         | List of wrong methods (0.5 off each) |                |
| 2. List the sequence of methods involved in depositing money into an account. | menu()  
run()  
deposit()  
findCustomer()  
<<create>>aTransaction=new DepositTransaction(...)  
execute()  
deposit(...)  
getBalance() | 1  
1  
2  
2  
2  
2  
2 |
|                                                                         | List of wrong methods (0.5 off each) |                |
| 3. a) List the class/classes associated with Account class.            | Account ----> Customers  
Transactions ----> Account  
Application ----> Account  
savings or checking class | 5  
5  
0  
0 |
|                                                                         | List of wrong classes (0.5 off each) |                |
| 3. b) Describe the nature of the relationship among those classes (i.e.1 to 1, 1 to Many, etc.). | each account contain 0 or multiple transaction: 1 to 0…*  
application contains 0 or multiple accounts: 1 to 0…* | 2  
0 |
|                                                                         | Application | 5 |
|                                                                         | List of wrong classes (0.5 off each) |                |
| 4. Name the class contains hash tables to save the customer and account objects. | menu()  
run() | 1  
1 |
<table>
<thead>
<tr>
<th>Question</th>
<th>Methods/Classes</th>
<th>Offs</th>
</tr>
</thead>
<tbody>
<tr>
<td>displaying a customer.</td>
<td>DisplayCustomer( )</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>findCustomer( )</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>writeCustomer(… )</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>List of wrong methods (0.5 off each)</td>
<td></td>
</tr>
<tr>
<td>6. There are 6 main options listed in the system description. Which of</td>
<td>CreateNewAccount option 2)</td>
<td>4</td>
</tr>
<tr>
<td>those options would retrieve customer objects from the customerTable?</td>
<td>DisplayCustomer option 5)</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>List of wrong option (0.5 off each)</td>
<td></td>
</tr>
<tr>
<td>7. Name the class responsible for creating a new transaction object.</td>
<td>Application( )</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>List of wrong classes (0.5 off each)</td>
<td></td>
</tr>
<tr>
<td>8. List all the attributes that describe a business customer object.</td>
<td>CustomerID</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>CustomerName</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>CustomerAddress</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>OrgName</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>OrgID</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>OrgAddress</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>List of wrong attributes (0.5 off each)</td>
<td></td>
</tr>
<tr>
<td>9. Describe the hierarchy of customers, i.e., different customer types</td>
<td>Account</td>
<td>2</td>
</tr>
<tr>
<td>and how they are related.</td>
<td>Business</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>Personal</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>Explanations (inheritance, super/sub class, child/parent)</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>List of wrong classes (0.5 off each)</td>
<td></td>
</tr>
<tr>
<td>10. a) List the class/classes associated with the Transaction class.</td>
<td>Transaction --&gt; Account</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>Application----- Transaction</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>List of wrong classes (0.5 off each)</td>
<td></td>
</tr>
<tr>
<td>10. b) Describe the nature of the relationship among those classes. (i.e.</td>
<td>Each account contain 0 or multiple transaction: 1 to 0…*</td>
<td>2</td>
</tr>
<tr>
<td>1 to 1, 1 to Many, etc.)</td>
<td>Behavioral Questions</td>
<td>44</td>
</tr>
<tr>
<td></td>
<td>Structure Questions</td>
<td>44</td>
</tr>
</tbody>
</table>
Appendix I

Task 1: Structural Task—Coding Rubrics
<table>
<thead>
<tr>
<th>I: Evaluate creating savings and checking account</th>
<th>Perfect</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Program asks users to open a savings or checking account</td>
<td></td>
</tr>
<tr>
<td>1.1 createAccount( ) in Application Class</td>
<td></td>
</tr>
<tr>
<td>A Evaluate type of account to be created (while, if/else sentence)</td>
<td>3</td>
</tr>
<tr>
<td>Option a: non-inherited classes</td>
<td></td>
</tr>
<tr>
<td>Created two account objects</td>
<td>6</td>
</tr>
<tr>
<td>Option b: inherited Classes</td>
<td></td>
</tr>
<tr>
<td>Created savings and checking object separately</td>
<td>6</td>
</tr>
<tr>
<td>B Enter initial balance</td>
<td>2</td>
</tr>
<tr>
<td>C evaluate initial checking balance&gt;=250</td>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>2 Program Creates and Display Savings/Checking Account</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Option a: non-inherited class</td>
<td></td>
</tr>
<tr>
<td>2a.1 Modify Account Class constructor</td>
<td></td>
</tr>
<tr>
<td>public Account(Customer aCustomer, double initialBalance, string type)</td>
<td></td>
</tr>
<tr>
<td>{ assignments to each parameter }</td>
<td>5</td>
</tr>
<tr>
<td>2a.2 Add &quot;Account Type&quot; to account display (Application Class)</td>
<td>2</td>
</tr>
<tr>
<td>Add &quot;Account Type&quot; to Withdraw/Deposit display (Transaction class)</td>
<td>4</td>
</tr>
<tr>
<td>evaluate checking final balance&gt;=250 (Application Class)</td>
<td>4</td>
</tr>
</tbody>
</table>

Option b: inherited savings/checking class were created

<p>| 2b.1 creation of savings class | 3 |
| 2b.2 savings class inherits from Account | 2 |
| 2b.3 Constructor | |
| public Savings(Customer aCustomer, double initialBalance) | |
| { super(aCustomer, initialBalance); } | 2 |
| 2b.4 withdraw( ) | |
| add &quot;account type: savings&quot; to withdraw transaction display | 2 |
| 2b.5 toString( ) | |
| &quot;Account type: savings&quot; + super.toString | 2 |
| 2b.6 creation of checking class | 3 |
| 2b.7 checking class inherits from Account | 2 |
| 2b.8 Constructor | |
| public Checking(Customer aCustomer, double initialBalance) | |
| { super(aCustomer, initialBalance); } | 2 |
| 2b.9 withdraw( ) | |
| add &quot;account type: savings&quot; to withdraw transaction display | 2 |</p>
<table>
<thead>
<tr>
<th>Code Grading Rubrics Cont.</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>evaluate final balance &gt;= $250</td>
<td>4</td>
</tr>
<tr>
<td>2b.10 toString()</td>
<td>2</td>
</tr>
<tr>
<td>“Account type: savings” + super.toString</td>
<td></td>
</tr>
</tbody>
</table>

II: Evaluate Display Transaction

<table>
<thead>
<tr>
<th>3</th>
<th>Program asks users to choose Display Transaction option</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>menu() in Application class</td>
</tr>
<tr>
<td>a</td>
<td>7: Display Transaction</td>
</tr>
<tr>
<td>3.2</td>
<td>Run() in Application class</td>
</tr>
<tr>
<td>b</td>
<td>final int QUIT = 8;</td>
</tr>
<tr>
<td>c</td>
<td>case 7: DisplayTransaction();</td>
</tr>
<tr>
<td>3.3</td>
<td>Create DisplayTransaction() in Application Class</td>
</tr>
<tr>
<td>a</td>
<td>Ask agent to input transaction number</td>
</tr>
<tr>
<td>b</td>
<td>verify existing transaction numbers in Hashtable or ArrayList</td>
</tr>
<tr>
<td>c</td>
<td>Display following Transaction information</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>4</th>
<th>Save transactions to hashtable or arraylist in Application class</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>System creates spaces to save transactions</td>
</tr>
<tr>
<td>Option1: created hashtable in Application class</td>
<td>8</td>
</tr>
<tr>
<td>Option2: create arraylist in Application class</td>
<td>4</td>
</tr>
<tr>
<td>4.2</td>
<td>Save transactions in Harshtable or arraylist</td>
</tr>
<tr>
<td>e.g. save to HarshTable</td>
<td></td>
</tr>
<tr>
<td>transactionTable.put(aTransaction.getTransactionNumber, aTransaction);</td>
<td>3</td>
</tr>
<tr>
<td>create saveTransaction(...) in Application class</td>
<td>2</td>
</tr>
<tr>
<td>4.3</td>
<td>save transaction in withdraw()</td>
</tr>
<tr>
<td>4.4</td>
<td>save transaction in deposit()</td>
</tr>
<tr>
<td>5</td>
<td>Run the code</td>
</tr>
</tbody>
</table>
Appendix J

Task 2: Behavioral Task—Description
System Descriptions

ABC Bank is in the process of upgrading its banking application. Currently, the system allows bank agents to perform following actions:

Action 1: Create New Customer
The system allows bank agents to create new customers. The bank agent assigns each new customer with a customer ID.

The system records customer information with all associated accounts information.

Action 2: Create New Account
The system allows bank agents to create two types of new accounts—checking and savings. Each customer may have more than one account. The system assigns each new account with an account number.

Action 3: Deposit Transaction
Deposit transaction can be performed on an account. The system assigns each new Deposit transaction with a transaction number.

Action 4: Withdrawal Transaction
Withdrawal transaction can be performed on an account. The system assigns each new withdrawal transaction with a transaction number.

Action 5: Display Customer Information
The system displays customer information using customer ID.

Action 6: Display Account Information
The system displays account information with associated customer information using account number.
Appendix K

Task 2: Behavioral Task—Modification Requirements
Enhancement Task Description

In order to be competitive in the marketplace, ABC Bank is trying to add some new capabilities and features to their application. The new requirements are listed below:

1. The system will allow a new action. Action 7: Transfer Transaction. The new option will allow bank agent to transfer money between accounts (between savings, checking or savings and checking). In addition, the maximum transfer amount per transaction is $2000.00.
2. System will display customer information with all associated accounts (account number and balance).
Appendix L

Task 2: Behavioral Task—UML Diagrams
Class Diagram

Create New Customer

Sequence Diagrams

Application

Customer

Account

Transaction

Settings

Checking

Deposit Transaction

Withdraw Transaction

Bank Agent

1: CreateCustomer()

Ask to enter customer name, ID and address

2: <<Create>> (id, name, address)

Customer

3: saveCustomer(aCustomer)

save customer information in Hashtable
Create New Account (Savings or Checking)

1. CreateAccount()

2. $aCustomer = findCustomer()$

Ask to enter customer ID

3. $<<Create>> (aCustomer, InitialBalance) 

anAccount : Account

4. $saveAccount(anAccount)$

Save account information in HasTable

Deposit Transaction

1. deposit()

2. $anAccount = findAccount()$

Ask to enter account number

3. $<<Create>> (anAccount, "deposit", amount) 

$<<Abstract>> 

anTransaction : Transaction

4. $InitialBalance = getBalance()$

5. execute()
Appendix M

Task 2: Behavioral Task—Comprehension Questions and Rubrics
<table>
<thead>
<tr>
<th>Question</th>
<th>Answer</th>
<th>Perfect Points</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. List the sequence of methods involved in the creation of a savings account object.</td>
<td>menu()</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>run()</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>createAccount()</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>findCustomer()</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>&lt;&lt;create&gt;&gt;anAccount = new Savings(...)</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>addAccount(...)</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>saveAccount(...)</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>Other methods:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>accountsTable.put(...)</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>setCustomer(...)</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>getResponse()</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>List of wrong methods (0.5 off each)</td>
<td></td>
</tr>
<tr>
<td>2. List the sequence of methods involved in depositing money into an account.</td>
<td>menu()</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>run()</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>deposit()</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>findCustomer()</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>&lt;&lt;create&gt;&gt;aTransaction=new DepositTransaction(...)</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>getBalance()</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>execute()</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>deposit(...)</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>getBalance()</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>SetFinalBalance(...)</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>List of wrong methods (0.5 off each)</td>
<td></td>
</tr>
<tr>
<td>3. a) List the class/classes associated with Account class.</td>
<td>Account ---- Customers</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>Transactions ----&gt; Account</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>Application ----&gt; Account</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>savings or checking class</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>List of wrong classes (0.5 off each)</td>
<td></td>
</tr>
<tr>
<td>3. b) Describe the nature of the relationship among those classes (i.e.1 to 1, 1 to Many, etc.).</td>
<td>each customer could have 0 or multiple accounts 1 to 0…*</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>each account only belongs to one customer: 1 to 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>each account contain 0 or multiple transaction: 1 to 0…*</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>application contains 0 or multiple accounts: 1 to 0…*</td>
<td>2</td>
</tr>
<tr>
<td>4. Name the class contains hash tables to save the customer and account objects.</td>
<td>Application</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>List of wrong classes (0.5 off each)</td>
<td></td>
</tr>
<tr>
<td>5. List the sequence of methods involved in displaying a customer.</td>
<td>menu()</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>run()</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>DisplayCustomer()</td>
<td>2</td>
</tr>
<tr>
<td>Question</td>
<td>Points</td>
<td></td>
</tr>
<tr>
<td>-------------------------------------------------------------------------</td>
<td>--------</td>
<td></td>
</tr>
<tr>
<td>6. There are 6 main options listed in the system description. Which of</td>
<td></td>
<td></td>
</tr>
<tr>
<td>those options would retrieve customer objects from the customerTable?</td>
<td></td>
<td></td>
</tr>
<tr>
<td>List of wrong methods (0.5 off each)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7. Name the class responsible for creating a new transaction object.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>List of wrong classes (0.5 off each)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8. List all the attributes that describe a checking account.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>List of wrong attributes (0.5 off each)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9. Describe the hierarchy of account, i.e., different account types and</td>
<td></td>
<td></td>
</tr>
<tr>
<td>how they are related.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>List of wrong classes (0.5 off each)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10. a) List the class/classes associated with the Transaction class.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>List of wrong classes (0.5 off each)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10. b) Describe the nature of the relationship among those classes (i.e.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 to 1, 1 to Many, etc.).</td>
<td></td>
<td></td>
</tr>
<tr>
<td>List of wrong classes (0.5 off each)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Behavioral Questions</td>
<td>48</td>
<td></td>
</tr>
<tr>
<td>Structure Questions</td>
<td>48</td>
<td></td>
</tr>
</tbody>
</table>
Appendix N

Task 2: Behavioral Task—Coding Rubrics
<table>
<thead>
<tr>
<th>1</th>
<th>Program asks users to choose transfer option</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>menu() in Application Class</td>
</tr>
<tr>
<td>a</td>
<td>7: Transfer Transaction</td>
</tr>
<tr>
<td>1.2</td>
<td>Run() in Application Class</td>
</tr>
<tr>
<td>b</td>
<td>final int QUIT = 8;</td>
</tr>
<tr>
<td>c</td>
<td>case 7: Transfer();</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>2</th>
<th>Program requests user for the source/target number</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Transfer() in Application Class</td>
</tr>
<tr>
<td>a</td>
<td>Ask agent to input source/target account number</td>
</tr>
<tr>
<td>b</td>
<td>use findAccount() to verify accounts numbers</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>3</th>
<th>Program request amount to be transferred</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Transfer() in Application Class</td>
</tr>
<tr>
<td>a</td>
<td>Ask agent to input the transfer amount</td>
</tr>
<tr>
<td>b</td>
<td>verify the transfer amount &lt;$2000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>4</th>
<th>Program creates a transfer transaction</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Transfer() in Application Class</td>
</tr>
<tr>
<td>Option a:</td>
<td>Inherited Classes - Created a TransferTransaction object</td>
</tr>
<tr>
<td>Option b,</td>
<td>separated class: Created new withdrawTransaction and depositTransaction object</td>
</tr>
<tr>
<td>Option c,</td>
<td>non-separated class: Call withdraw() and deposit()</td>
</tr>
<tr>
<td>Option d:</td>
<td>used setBalance() in Account class</td>
</tr>
<tr>
<td>Option e:</td>
<td>implement transferTo() in Account class</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>5</th>
<th>Display following Transfer transaction information. Changes in Application Class, Transaction Class or TransferTransaction Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>Transaction number</td>
</tr>
<tr>
<td>b</td>
<td>Transaction Date</td>
</tr>
<tr>
<td>c</td>
<td>Type of Transaction</td>
</tr>
<tr>
<td>d</td>
<td>Source account number</td>
</tr>
<tr>
<td>d</td>
<td>Beginning balance of source account</td>
</tr>
<tr>
<td>e</td>
<td>Ending balance of source account</td>
</tr>
<tr>
<td>f</td>
<td>Target account number</td>
</tr>
<tr>
<td>g</td>
<td>Beginning balance of target account</td>
</tr>
<tr>
<td>h</td>
<td>Ending balance of target account</td>
</tr>
</tbody>
</table>

| 6 | Transfer money from one account to another |
**Option a: A TransferTransaction class was created**

<table>
<thead>
<tr>
<th>6a.1</th>
<th>creation</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>6a.2</td>
<td>TransferTransaction class inherits from Transaction</td>
<td>3</td>
</tr>
<tr>
<td>6a.3</td>
<td>Constructor</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>public TransferTransaction(Account outAccount, Account inAccount, double amount)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>{ Assignments to each parameter }</td>
<td></td>
</tr>
<tr>
<td>6a.4</td>
<td>execute ( )</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>sourceAccount.withdraw(amount);</td>
<td></td>
</tr>
<tr>
<td></td>
<td>sourceAccountfinalBalance = sourceAccount.getBalance( );</td>
<td></td>
</tr>
<tr>
<td></td>
<td>targetAccount.deposit(amount);</td>
<td></td>
</tr>
<tr>
<td></td>
<td>targetAccountfinalBalance = targetAccount.getBalance( );</td>
<td></td>
</tr>
<tr>
<td>6a.5</td>
<td>toString( )</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>display all information in section 5</td>
<td></td>
</tr>
</tbody>
</table>

**Option b: depositTransaction and withdrawTransaction classes were modified**

<table>
<thead>
<tr>
<th>6b.1</th>
<th>toString( ) in depositTransaction classes</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>display all information in section 5</td>
<td></td>
</tr>
<tr>
<td></td>
<td>toString( ) in withdrawTransaction classes</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>display all information in section 5</td>
<td></td>
</tr>
</tbody>
</table>

**II: Evaluate Display Customer & Accounts**

<table>
<thead>
<tr>
<th>7</th>
<th>read accountList&lt;array&gt; in Customer class</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Use loop sentence to read accounts in arrayList</td>
<td></td>
</tr>
<tr>
<td></td>
<td>e.g. (Account account : accountsList)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>{ accountInfo += account; }</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Implement loop logic in Customer class.</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>option1: Created a new method to implement loop logic in Customer class</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>option2: Implement loop logic in toString( ) in Customer class</td>
<td>2</td>
</tr>
<tr>
<td>9</td>
<td>Run the code</td>
<td>5</td>
</tr>
</tbody>
</table>
References


Welch, B.L. *On the comparison of several mean values: An alternative approach* Biometrika, 1951.
Biographical Information

Lulu Zhang received her doctorate in Business Administration with a major in Information Systems from the University of Texas at Arlington. She holds Master of Science in Engineering and Management of Information Systems from Royal Institute of Technology, Sweden. She received her Bachelor of Engineering in Telecommunication Engineering from Beijing University of Posts & Telecommunications, China.

She has over five years of industry experience in project management positions. Her current research interests include project management, software development, cognitive behavior, social networks, IS management, and research methodologies.