TOWARD UNDERSTANDING THE COGNITIVE PROCESSES OF SOFTWARE DESIGN IN NOVICE PROGRAMMERS

A Dissertation in
Instructional Systems
by
Kuo-Chuan Yeh

© 2009 Kuo-Chuan Yeh

Submitted in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

August 2009
The thesis of Kuo-Chuan Yeh was reviewed and approved* by the following:

Kyle Peck  
Professor of Education and Associate Dean of Outreach  
Dissertation Co-Advisor  
Chair of Committee

Christopher Hoadley  
Associate Professor of Educational Communications and Technology  
New York University  
Dissertation Co-Advisor

John D. Popp  
Affiliate Assistant Professor of  
Instructional Systems

Frederico Fonseca  
Associate Professor of  
Information Sciences and Technology

Edgar I. Farmer  
Professor of Education  
Department Head, Learning and Performance Systems

*Signatures are on file in the Graduate School
ABSTRACT

This study provides insights with regard to the types of cognitive processes that are involved in the formation of mental models and the way those models change over the course of a semester in novice programmers doing a design task. Eight novice programmers participated in this study for three distinct software design sessions, using the same design scenario, at different times during the semester. Verbal protocols were collected and analyzed by segmenting them into a series of episodes, which were then categorized as one of five types of cognitive processes: problem comprehension, decomposition, structuring, mental simulation, and evaluation. Episodes that did not directly relate to the design problem are categorized as a non-design type of activity. Both the types of episodes and the finished design documents were analyzed to project what type of mental models the novice programmers have, how they use their mental models in a design process, and how they restructure their mental models.

Two types of mental models were used by the novice programmers in this study. One was a UI-based mental model, which is the most common mental model formed by the novice programmers. It is an easy to use, action-oriented mental model created by the owner based on his experience interacting with the system to be designed. It tends to tax designers’ mental simulation ability, but is simple and shallow. The second type of mental model, which is also used by novice programmers, although not as frequently, is a programming-based mental model in which novice programmers incorporate programming concepts and symbolic representation into their design. Because the activity of programming is detail-oriented, and procedurally-driven, designers using this model tend to focus on decomposition and structuring.

This study also finds that the prototypical mental model the novices create in the beginning of a design session does not have enough detail and is not executable. They often change and reconstruct their mental models during the design process. The inability to create an
executable mental model is probably one of the reasons that they demonstrate a more opportunistic design pattern than a systematic one (i.e., a top-down strategy). As a result, for novice programmers, the problem solving process becomes the consolidation of mental models.
# TABLE OF CONTENTS

LIST OF FIGURES ................................................................................................................. vii

LIST OF TABLES ................................................................................................................... x

ACKNOWLEDGEMENTS ..................................................................................................... xii

Chapter 1 INTRODUCTION ................................................................................................... 1

  Research Questions .......................................................................................................... 5
  Significance ...................................................................................................................... 6
  Limitations ....................................................................................................................... 7

Chapter 2 REVIEW OF LITERATURE .................................................................................. 8

  Human Cognitive Architecture ..................................................................................... 8
  Problem Solving ................................................................................................................. 10
    Production System ........................................................................................................ 11
  Psychology of Software Development ............................................................................. 12
    Knowledge ...................................................................................................................... 12
    Strategy ........................................................................................................................ 18
  Mental Models ................................................................................................................. 23
  Summary .......................................................................................................................... 26

Chapter 3 METHODOLOGY .................................................................................................. 27

  Participants ....................................................................................................................... 28
  Instruments ....................................................................................................................... 29
  Procedures ........................................................................................................................ 31
  Pilot Study ......................................................................................................................... 34
    Participants .................................................................................................................... 34
    Procedures .................................................................................................................... 35
    Findings ........................................................................................................................ 36
  Data Analysis ................................................................................................................... 37
    Coding themes ............................................................................................................. 39
    Reliability ..................................................................................................................... 43
  Summary .......................................................................................................................... 45

Chapter 4 RESULTS OF THE RESEARCH ........................................................................... 47

  Experts’ Data ..................................................................................................................... 47
  Demographic Data ............................................................................................................. 47
  Design Behavior ............................................................................................................... 48
  Cognitive Processes .......................................................................................................... 50
  Qualitative Analysis ......................................................................................................... 53

  Novices’ Data ....................................................................................................................... 55
  Benjamin ........................................................................................................................ 55
  Caleb .................................................................................................................................. 63
LIST OF FIGURES

Figure 2.1 Human cognitive architecture (adapted from Atkinson & Shiffrin, 1986) .......... 8

Figure 4.1 Comparison of experts’ design cognitive processes by the number of episodes... 51

Figure 4.2 Comparison of experts’ design cognitive processes by the percentage of episodes ........................................................................................................................... 52

Figure 4.3 A scatter plot representing E1’s chronological design processes .................... 52

Figure 4.4 A scatter plot representing E2’s chronological design processes .................... 53

Figure 4.5 A scatter plot representing Benjamin’s chronological design processes in design session I ........................................................................................................... 57

Figure 4.6 A scatter plot representing Benjamin’s chronological design processes in design session II ............................................................................................................ 59

Figure 4.7 A scatter plot representing Benjamin’s chronological design processes in design session III ........................................................................................................... 60

Figure 4.8 Comparing Benjamin’s three design sessions by the number of design-related episodes ................................................................................................................... 62

Figure 4.9 Comparing Benjamin’s three design sessions by the percentage of design-related episodes ................................................................................................................... 62

Figure 4.10 A scatter plot representing Caleb’s chronological design processes in design session I .................................................................................................................. 65

Figure 4.11 A scatter plot representing Caleb’s chronological design processes in design session II .................................................................................................................. 67

Figure 4.12 A scatter plot representing Caleb’s chronological design processes in design session III .................................................................................................................. 68

Figure 4.13 Comparing Caleb’s three design sessions by the number of design-related episodes ....................................................................................................................... 69

Figure 4.14 Comparing Caleb’s three design sessions by the percentage of design-related episodes ....................................................................................................................... 70

Figure 4.15 A scatter plot representing Jedediah’s chronological design processes in design session I ............................................................................................................... 73

Figure 4.16 A scatter plot representing Jedediah’s chronological design processes in design session II ............................................................................................................... 75
Figure 4.17 A scatter plot representing Jedediah’s chronological design processes in design session III. ............................................................................................................. 77

Figure 4.18 Comparing Jedediah’s three design sessions by the number of design-related episodes. ........................................................................................................................... 78

Figure 4.19 Comparing Jedediah’s three design sessions by the percentage of design-related episodes. ........................................................................................................................... 78

Figure 4.20 A scatter plot representing David’s chronological design processes in design session I. ........................................................................................................................... 80

Figure 4.21 A scatter plot representing David’s chronological design processes in design session II. ........................................................................................................................... 81

Figure 4.22 A scatter plot representing David’s chronological design processes in design session III. ........................................................................................................................... 84

Figure 4.23 Comparing David’s three design sessions by the number of design-related episodes. ........................................................................................................................... 85

Figure 4.24 Comparing David’s three design sessions by the percentage of design-related episodes. ........................................................................................................................... 85

Figure 4.25 A scatter plot representing Lazarus’s chronological design processes in design session I. ........................................................................................................................... 88

Figure 4.26 A scatter plot representing Lazarus’s chronological design processes in design session II. ........................................................................................................................... 89

Figure 4.27 A scatter plot representing Lazarus’s chronological design processes in design session III. ........................................................................................................................... 91

Figure 4.28 Comparing Lazarus’s three design sessions by the number of design-related episodes. ........................................................................................................................... 92

Figure 4.29 Comparing Lazarus’s three design sessions by the percentage of design-related episodes. ........................................................................................................................... 92

Figure 4.30 Mary’s first visual representation of the online library management system in design session I. ........................................................................................................................... 94

Figure 4.31 A scatter plot representing Mary’s chronological design processes in design session I. ........................................................................................................................... 95

Figure 4.32 A scatter plot representing Mary’s chronological design processes in design session II. ........................................................................................................................... 96

Figure 4.33 A scatter plot representing Mary’s chronological design processes in design session III. ........................................................................................................................... 98
Figure 4.34 Comparing Mary’s three design sessions by the number of design-related episodes. ................................................................. 99

Figure 4.35 Comparing Mary’s three design sessions by the percentage of design-related episodes. ................................................................. 99

Figure 4.36 A scatter plot representing Matthew’s chronological design processes in design session I. ..................................................... 101

Figure 4.37 A scatter plot representing Matthew’s chronological design processes in design session II. ..................................................... 103

Figure 4.38 A scatter plot representing Matthew’s chronological design processes in design session III. .................................................. 105

Figure 4.39 Comparing Matthew’s three design sessions by the number of design-related episodes. ................................................................. 106

Figure 4.40 Comparing Matthew’s three design sessions by the percentage of design-related episodes. ................................................................. 106

Figure 4.41 A scatter plot representing Zeke’s chronological design processes in design session I. ......................................................... 107

Figure 4.42 A scatter plot representing Zeke’s chronological design processes in design session II. ......................................................... 110

Figure 4.43 A scatter plot representing Zeke’s chronological design processes in design session III. ......................................................... 111

Figure 4.44 Comparing Zeke’s three design sessions by the number of design-related episodes. ................................................................. 113

Figure 4.45 Comparing Zeke’s three design sessions by the percentage of design-related episodes. ................................................................. 113
LIST OF TABLES

Table 3.1 Ambiguity Examples and Solutions.........................................................................43
Table 4.1 Cognitive Process Frequency by Expert E1.............................................................50
Table 4.2 Cognitive Process Frequency by Expert E2.............................................................51
Table 4.3 Cognitive Process Frequency by Benjamin in Design Session I .............................56
Table 4.4 Cognitive Process Frequency by Benjamin in Design Session II ............................58
Table 4.5 Cognitive Process Frequency by Benjamin in Design Session III .........................60
Table 4.6 Cognitive Process Frequency by Caleb in Design Session I .................................64
Table 4.7 Cognitive Process Frequency by Caleb in Design Session II .................................66
Table 4.8 Cognitive Process Frequency by Caleb in Design Session III ...............................67
Table 4.9 Cognitive Process Frequency by Jedediah in Design Session I ..............................72
Table 4.10 Cognitive Process Frequency by Jedediah in Design Session II ............................74
Table 4.11 Cognitive Process Frequency by Jedediah in Design Session III ..........................76
Table 4.12 Cognitive Process Frequency by David in Design Session I ...............................79
Table 4.13 Cognitive Process Frequency by David in Design Session II ...............................81
Table 4.14 Cognitive Process Frequency by David in Design Session III .............................83
Table 4.15 Cognitive Process Frequency by Lazarus Design Session I .................................87
Table 4.16 Cognitive Process Frequency by Lazarus Design Session II ...............................89
Table 4.17 Cognitive Process Frequency by Lazarus Design Session III ..............................90
Table 4.18 Cognitive Process Frequency by Mary Design Session I .................................95
Table 4.19 Cognitive Process Frequency by Mary Design Session II .................................96
Table 4.20 Cognitive Process Frequency by Mary in Design Session III .............................97
Table 4.21 Cognitive Process Frequency by Matthew in Design Session I ...........................100
Table 4.22 Cognitive Process Frequency by Matthew in Design Session II ...........................102
Table 4.23 Cognitive Process Frequency by Matthew in Design Session III ..........................104
Table 4.24 Cognitive Process Frequency by Zeke in Design Session I ........................................ 107
Table 4.25 Cognitive Process Frequency by Zeke in Design Session II ...................................... 110
Table 4.26 Cognitive Process Frequency by Zeke in Design Session III .................................... 111
ACKNOWLEDGEMENTS

The Lord Jesus has been with me in every step of this long, unexpected journey. I hope by this work HIS name is glorified.

I want to acknowledge the people who have helped and supported me throughout the past six years.

First, I want to give the most sincere appreciation to every person on my thesis examining committee for all their intelligent comments and encouraging support. Dr. Kyle Peck is the one who brought me into and gave advice to help me get through the Instructional Systems program. Without him, I would not be able to study science from a more humanistic side. Dr. Christopher Hoadley is a great motivator and never gave up on me. He was willing to and capable of nurturing my academic and personal needs. Dr. Dave Popp helped me to understand the U.S. K-12 educational system and has always made himself available to answer my many questions. Dr. Fred Fonseca allowed me to sit in his class for one year and taught me how to be persistent and look for details.

Second, I also want to give every credit to my family—my parents, my wife Rose, and my two lovely daughters, Catherine and Elizabeth. They are my motivation and my energy. Their presence moves me forward and their smiles ease my stress. This work would never exist without them in my life.

Third, I want to thank everyone in State College Chinese Alliance Church for they have remembered me in their prayers and gave me guidance to keep me in God’s way. It is my blessing to walk with them. Through them, I see God’s kingdom.

Fourth, I want to express my gratitude to my friends, inside and outside of the Penn State community. Their consultation, comfort, and love helped me through some doubts and difficult
situations. I wish to thank Dr. Noela Haughton, Dr. Khusro Kidwai, Dr. Charlie Cox, Dr. Wei-Fan Chen, Yu-Chang Hsu, and Yu-Hei Ching for their valuable comments and suggestions.
Chapter 1

INTRODUCTION

“The most expensive errors to correct in a software development project are those made during high-level design. Yet, high-level design has rarely been empirically studied and is not currently well supported by software tools and environments.” (Guindon, 1989, p. 727)

Computer programming and software design can be seen as two different types of cognitive activities which are both essential to software development (Guindon, 1990). To learn computer programming is to acquire syntactical and semantic knowledge of specific programming languages, and to learn software design is to apply different types of knowledge to various problem solving situations (Jeffries, Turner, Polson, & Atwood, 1981). According to these definitions, programming knowledge relies heavily on the language itself, whereas global design knowledge can be generalized, to a certain extent, across domains (Goel & Pirolli, 1992). A software design problem is quintessentially an ill-structured or ill-defined problem, similar to many other problem solving situations in different domains. Therefore, borrowing theoretical frameworks from human problem solving seems to be a legitimate strategy for research studies in the cognition of software design. This study will examine cognitive processes and mental models in software design through the perspective of problem solving based on cognitive psychology theories.

Software design can mean different things to different people. Sometimes it simply means the action of generating or writing programs. It can also be used to refer to solving a complex, real-life problems with software. In computer programming courses, people refer to the action of programming as “designing software”; in software engineering courses, people refer to the process of recursively decomposing the problem statement into smaller components until they are
manageable and can achieve the overall goal by attaining those sub-components as “software
design”. Simon (1984) describes design as the desire originating from dissatisfaction with the
current situation and the resulting attempt to make things the way they ought to be. So, design is
a motivation to fulfill a need, and the science of design is to create artifacts, good ones. When we
switch our attention to software development, software design is described as “the process of
translating a set of task requirements (functional specifications) into a structured description of a
computer program that will perform the task” (Jeffries et al., 1981). Although people have
different ideas of what software design should be, no one can deny its bundled relationship with
programming, whereas programming activity rarely can be conducted without design.

Although design and programming are intertwined, software design is psychologically
different from computer programming because computer programming requires knowledge that is
stored in programming schemas. The content of a programming schema includes syntactical and
semantic knowledge, programming plans, templates, examples, etc. (Anderson, Farrell, & Sauers,
1984; Davies, 1990; Gilmore & Green, 1988; Pirolli & Anderson, 1985; Rist, 1986, 1989;
Soloway & Ehrlich, 1984). As a result, the programming schemas are sensitive to the constructs
of the programming language (Gilmore & Green, 1988) and to the learning processes of
programming (Davies, 1991a). On the other hand, software design usually calls for schemas from
multiple domains. Software design requires abstract knowledge and design process from design
schemas, which are domain independent (Jeffries et al., 1981). Therefore, studies about the
psychology of software design should be addressed at least somewhat independently.

Why do we want to focus on software design? If the acquisition of design knowledge is
tangled with experience as Adelson and Soloway (1985) would argue, can we hope that students
will gain design-related knowledge through formally and informally learning to program? Doing
so is asking for trouble, however, because learning computer programming in its own right is a
percent dropout rate in an introductory computer programming course (CS1) at Helsinki University of Technology. One of the difficulties for CS1 students is “managing extensive unity: for some students, the size of the project work was difficult to handle” (2006, p. 103), which means for some students, the programming project is too complex. Software design knowledge and skills provide a solution to control the complexity of a project. Because what makes programming difficult is not to memorize syntax and semantics, rather it lies in the design aspect of a problem (Soloway, 1986; Spohrer & Soloway, 1986a, 1986b).

Unfortunately the acquisition of programming and design knowledge are usually perceived as a sequential process where computer programming is taught before software design. If indeed there is a global design knowledge that applies across domains (Goel & Pirolli, 1992), why does the curriculum of computer science education place computer programming ahead of software design (The Joint Task Force on Computing Curricula IEEE/ACM, 2004)? Maybe by teaching basic design knowledge along with syntactic and semantic knowledge, the level of frustration students experience can be reduced, along with the dropout rate. While the actual reason may be a myth, the fact that software design postulates cross-disciplinary knowledge and remains a complex cognitive task (Goel & Pirolli, 1992; Jeffries et al., 1981) may play a major role in the decision making process of the design of computer science education curriculum. However, the sequence is not set in stone. Researchers (Curtis, Krasner, & Iscoe, 1988; Guindon, 1989) argue that the longer we defer teaching design knowledge, the more costly it will become. Not only is the cost of fixing a design error enormous, but also the loss of human capital due to the fact that learning programming is an unmotivated task for novice programmers (Kinnunen & Malmi, 2006) and will create a worker shortage. Either one of these problems will damage the industry profoundly. A deeper understanding of the nature of software design and the development of design knowledge will put us in a better position to retain learners and improve computer science education. Archer (1969) proposed that there are two elements in a design process, a logical one
and a creative one. Without going into the debate about whether creativity is innate or learned, stronger logical elements will improve the design knowledge as a whole.

There is a consistent effort to make static, systematic software development models. The waterfall model of software development (Royce, 1970) is probably the most well-known and most-discussed software development model, and it has had profound influence on training programmers since its debut. In the early days, when the waterfall model was widely practiced, having access to computer systems was a privilege. Only a handful of people had access to computers, and sometimes they had to go through a cumbersome process to reserve computing time in advance. Machine time for both coding and debugging was costly, and a software development task had to be planned ahead so that it used as little machine time as possible. Furthermore, because software systems were complex, a project needed to be broken down into smaller subtasks to increase the reliability and coherence of the programs so that subtasks were manageable. The scope of software development projects also tended to be large and sophisticated, and often led by government agencies, research centers, or large companies. Accurate and mistake-free programs were paramount because the system’s failure might influence a large number of people (for example taxation system) or even put people’s lives in danger (for example some NASA projects). In this era, software design was very focused on a rigorous practice of decomposition and planning.

Fast forward to today: people use computer applications for all kinds of purposes, such as record keeping, number calculation, data analysis, and typing, just to name a few, in addition to the aforementioned large and complex projects. The focus of a software development model, therefore, is no longer primarily robust and error-free programs, because this is not always economically feasible. There are new programs being released everyday for personal and small business users who demand up-to-date features over stability. In order to remain competitive, small software companies cannot always afford to follow the waterfall model anymore.
Nowadays, many software developers frequently use the spiral model (Boehm & Hansen, 2001) and other agile development methods for software development, whether the software system is big or small. New programming language constructs and concepts, such as object-oriented programming, and tools, such as the highly adaptable integrated development environment (IDE), blossom and are spurred on by these paradigm changes. Although the content of what is being taught has been following in the footsteps of this change, the sequence of software development has not (The Joint Task Force on Computing Curricula IEEE/ACM, 2004).

In addition to sequencing, there are some other problems related to software design education. McCracken (2004) points out that among the scarce studies on software design, documentation on the progress of learning software design knowledge is almost nonexistent. Teaching without such knowledge is like driving in an unfamiliar neighborhood without directions. We might get to the destination, but it could take us a long time.

**Research Questions**

This study examines novice programmers’ mental models of software design and the cognitive processes that they use over time. Mayer (1989, p. 90) defined a mental model as “a metaphor consisting of components and operating rules which are analogous to the components and operating rules of the system.” So, for designers, a mental model is their sandbox in which they can operate and test hypotheses and alternatives. This is crucial for a design activity, but it does not operate by itself. A variety of cognitive processes are expected to cooperate with the mental model in a design process. Therefore, both the mental models and cognitive processes of novice programmers in a software design activity are to be identified and examined. Moreover, rather than taking a single snapshot of a novice programmer’s cognitive status, this study will also
discuss the development of mental models, giving software design activities to novice programmers at different times during a semester.

The research questions for this study are:

- What are the characteristics of the novice programmer’s software design mental model?
- What types of mental models do novice programmers use?
- What causes novice programmers’ software design mental models to change? And what are the changes?

**Significance**

Several studies (Adelson, 1981; Adelson, Littman, Ehrlich, Black, & Soloway, 1984; Winslow, 1996) have outlined the psychological differences between novice and expert programmers. Comparing the novice and the expert is a classical research design in human problem solving research. The rationale behind it is influenced by the view that there is an initial state and end state in any problem solving situation, and human problem solving is directional (toward the end state) and purposeful (to remove the obstacles between the initial and the end state). Research related to documenting and helping learners transition from the initial state to goal state in software design, begins by seeing how novices and experts differ in their problem solving, and then trying to help novices gain the tools and experiences needed to function more like experts.

This work is an early attempt to fill in the gap between the initial and the end state in software design expertise (as contrasted with programming expertise). In particular, the findings will direct the development of teaching software design and facilitate the transition from a novice to the next level.
Limitations

In the real world, most software development projects require a group of software designers and programmers, since most software development takes place in teams both in school and afterwards in professional practice. In such projects, the representations of individuals and shared representations of the group members (Pew & Mavor, 2007) influence the quality of the projects. The scope of this work is limited to the individual level; the influence of group collaborations and social interactions is beyond this work. Nevertheless, it can be a foundation for future study in those areas.

The conclusions were drawn from a single rater’s coding result. The absence of other blind raters makes the results vulnerable to bias. Duplicating this study may lead to different data sets and, hence, conclusions. In addition, the nature of this study is an explorative, descriptive study in an uncontrolled environment. The results of this study may not generalize to a broader population. Rather, it is meant to be an insightful and initial phase for more detailed, controlled quantitative studies.
Chapter 2

REVIEW OF LITERATURE

“...there is merit to the claim that much problem solving effort is directed at structuring problems, and only a fraction of it at solving problems once they are structured.” (Simon, 1973, p. 187)

Because this work will address software design issues from the perspective of cognitive psychology, I will first present a brief literature review on human cognitive architecture and problem solving. Then I will review recent research studies on software development, both programming and design, from the perspective of knowledge, strategy, and mental model.

**Human Cognitive Architecture**

![Diagram of Human Cognitive Architecture](Figure 2.1 Human cognitive architecture (adapted from Atkinson & Shiffrin, 1986).)

All human learning activities are related to a cognitive architecture that consists of sensory memory (SM), short-term memory (STM), long-term memory (LTM), and the partnerships they share (Atkinson & Shiffrin, 1968). Figure 2.1 presents the relationships among these three components. SM includes perception, hearing, taste, smell, and touch and is where information comes into STM. Of all the literature on learning-related cognition, perception is
most often discussed. Human perception is highly selective and adaptive to prior knowledge (Anderson, 1985; Driscoll, 2000). Through attention and pattern recognition, SM filters information and places it in STM.

Information getting through SM will first be stored in STM for a very short period of time, as its name implies. Peterson and Peterson (1959) conducted an empirical study and found that the retention for individual verbal items was about 18 seconds. Moreover, the amount of information can be contained in STM simultaneously is limited to seven plus/minus two (Miller, 1956). Chase and Ericsson (1981) argue that this amount could be increased with purposeful training. Another method of expanding STM is through chunking meaningful information, which is also introduced by Miller (1956). When information is organized into chunks, it does not need as much effort to be processed in STM as it does when it is stored independently.

Once information is learned, it is encoded in a structure, called a schema, through a process called schema acquisition, and stored in LTM. From the information-processing theorist’s perspective, learning is when new information is encoded and integrated into existing schema. Unlike STM, LTM has unlimited storage and retains information from several days to years, until it decays (Anderson, 1985; Kirschner, 2002). Before a solver can respond to a stimulus, existing knowledge must be retrieved from LTM into STM for processing. Therefore, STM is also called the working memory that is “holding only knowledge currently in use” (Anderson, 1985, p. 138).

Software design apparently requires information being retrieved from LTM into the limited capacity of STM. How does information-processing theory inform us about optimizing the problem solving process? Sweller (1988) suggests that the means-ends analysis type of problem solving activity requires a relatively large amount of cognitive processing capability and it prevents schema acquisition. In other words, some forms of problem solving may, in fact, interfere with learning. Therefore, to design learning for software design, and other domain
knowledge alike, one should consider the effect of cognitive load to minimize its negative impact on learning (Kirschner, 2002).

**Problem Solving**

Software design is in essence a problem solving exercise. Among many theoretical frameworks available for the research of psychological of software design, problem solving is one of the prominent ones (Détienne, 2001). To understand problem solving, we should first look at the type of problem that is to be solved. Simon (1973) discusses what differentiates ill-structured problems (ISP) from well-structured problems (WSP) and says that the dichotomy between ISP and WSP is a continuum with no clear boundary between them. We can only use a residual concept to understand ISP—if a problem is not a WSP, then it is an ISP. So, in what regard are we content that the boundary is vague? Simon said “there may be nothing other than the size of the knowledge base to distinguish ISPs from WSPs.” (1973, p. 197) He argues that we can divide an ISP into smaller problem spaces, and repeat the process until the smaller problem space is a WSP. If it is not, repeat the same process until it is. In the end, what differentiates ISP and WSP is the knowledge base of the problem solver, and a large portion of the problem solving effort resides in structuring. Using this analogy, the best way to approach a complex software design problem is to be able to divide the problem into smaller, more manageable parts that the designers can handle.

Problem solving knowledge has a strong influence on problem solving processes. Some problems, such as mutilated checkerboard problem, two-string problem, or the candle problem (Anderson, 1985), require a smaller amount of domain-specific knowledge. Problem representation is extremely important for these knowledge-lean problems, which are often referred as insight or intuition problems (Ohlsson, 1984). Another type of problem, such as those
found in mathematics, physics, medical diagnoses, or driving a car, demands rich domain-specific knowledge. Software design is one of these knowledge-rich types of problems (Adelson & Soloway, 1985).

To document the process involved in software design, a solid theoretical framework and a robust research method are needed. The most commonly used problem solving representation is a rule-based system called a production system; the most commonly used research methodology is through verbal protocol analysis. I will introduce production systems in the next section and verbal protocol analysis in the next chapter.

**Production System**

Newell and Simon (1972) describe human problem solving in a systematic and hierarchical method, often referred to as a production system. In such a system, knowledge is represented as a set of rules. Given a problem space which consists of all possible states and actions, solving a problem is in essence finding a path from the initial state to the goal state. The initial state is when a problem solver attempts the problem, and the goal state is when the system fulfills the requirements of problem descriptions. There can be several such paths that lead to the goal state, associated with different costs. Experts are those who can find a less expensive path.

The seminal work of Newell and Simon (1972) and the increasing calculation power of computers, made the simulation of intelligence through the implementation of the production system possible. This state-of-art representation of problems inspired other researchers and gave birth to projects of computational modeling, like ACT-R (Anderson & Lebiere, 1998) and SOAR (Newell, 1990), which, in turn, enable further human problem solving research (Lovett, 2002). Brooks (1977) implemented and tested a model for programming and concluded that top-down programming can only be used successfully in certain conditions. The merit of his work is that
researchers can create a model of problem solving, implement a corresponding computational model, and then simulate it. The simulation results can provide objective verification of whether the models mimic human problem solving.

**Psychology of Software Development**

In the following section, I will introduce literature that has contributed to the research in the psychology of software development (from both programming and design aspects). Three areas will be addressed: knowledge, strategy, and mental model.

**Knowledge**

For cognitive psychologists, schema is one of the fundamental constructs used to explain and predict learning. The most common metaphor is the network view of a schema where a schema consists of units of information and the relationships that connect those units. The units and the relationships change through learning and practice. Any anthropometry, behavior, cognition, or social cultural experience (Ritter & Churchill, 2008) could change a person’s schema through accommodation and/or assimilation (Davies, 1994). Moreover, schemas are organized differently for every person depending on the context where learning occurred.

McKeithen, Reitman, Rueter and Hirtle (1981) were inspired by expert programmers’ exceptional skills on recalling far more relevant information from their LTM memory than novice programmers could and conducted two empirical studies on whether the experts’ superior ability came from the possession of more relevant information or from better chunking and retrieving skills. In the first study, 53 participants were divided into three skill levels: 24 beginners, 23 intermediates, and 6 experts, based on their ALGOL W programming experience. A coherent or
scrambled version of 31 lines of a computer program was shown to the participants on an overhead projector for two minutes. Following each trial, in a three-minute recall period, the participants wrote what they could recall verbatim on a blank sheet. Their performance was measured by counting the number of correct lines where both the indentation and sequence of the line needed to be correct. Each participant did five trials without looking at previous attempts. They concluded in the first study that experienced programmers showed superior recall ability in the coherent version of the program; skilled programmers did not outperform beginners in the scrambled version of the program. Furthermore, expert programmers demonstrated chunking skills in the recall trials, by recalling lines that were composed of the same functionalities as the stimulus program.

In the second study, using the same grouping strategy as previous study, 22 participants, 8 experts, 6 intermediates and 8 beginners, were given 21 cards with ALGOL W reserved words on them. Each participant memorized the reserve words until they could recall all 21 words in two consecutive trials. Participants then tried to recall the 21 words in 25 trials, in which 21 trials were cued with one of the 21 words and the rest were not cued. The recall trials were video recorded to analyze the homogeneity of organization for all three skill levels. They concluded that programmers of different skill levels organize their knowledge differently. Beginners used mnemonic strategy to chunk reserve words by length and first letter, whereas experienced programmers chunked information by functional associations. Chase and Simon (1973) also observed chunking behavior in chess masters and proposed that experts had better chunking strategies than novices.

McKeithen, Reitman, Rueter and Hirtle’s studies (1981) are early attempts in applying schema theory in examining the differences in knowledge organization between different skill level programmers. There are two implications to this thesis work from their studies. Their
studies (1) consolidated the validity of applying schema theory in the psychology of software development, and (2) paved the road for other chunking strategy studies in software development.

Similarly, Adelson (1981) studied the chunking strategies that programmers used by examining the amount of program code the participants could recall. There were two types of participants in her study: novice programmers who were undergraduate students who had just completed a Polymorphic Programming Language (PPL) course and expert programmers who were instructors of PPL. They were all asked to watch a 16 line program written in PPL, one line at a time until all 16 lines were displayed. Each line appeared separately and randomly for 20 seconds on a computer screen and previous line was cleared before the following line was displayed, and then the participants were given eight minutes to recall as many items as they could. Subjects repeated this process nine times throughout the study. Quantitative data such as the number of items recalled and the size of chunking was collected and analyzed. Items that were recalled in less than 10 seconds preceding the previous ones were defined as current chunks. In this study, experts were able to recall significantly more than the novices and also form larger chunking sizes. Qualitative data was also collected to scrutinize the superior recall ability of the experts. Examining the scaling and clustering solutions of inter-item similarity indicated that novices organized programs by syntactic categories, whereas experts organized items semantically.

Soloway and Ehrlich (1984) extended the chunking strategy further by conducting empirical studies to examine an expert programmer’s programming knowledge and claimed that expert programmers used at least: (1) *programming plans* and (2) *rules of programming discourse* in computer program writing and comprehension. They referred to the programming plans as “generic program fragments that represent stereotypic action sequences in programming” and the rules of programming discourse as rules that “capture the conventions in programming and govern the composition of the plans into programs” (1984, p. 595). To test their hypothesis,
they devised both plan-like and nonplan-like program segments for participants in two activities: fill-in-the-blanks and verbatim recall. They assumed that expert programmers had a programming schema organized around programming plans which were readily available from LTM to STM, therefore, the experts would benefit from comprehending a plan-like program. On the other hand, programs that violate programming plan structures would take longer to comprehend and memorize. Therefore expert programmers should perform significantly better in the plan-like situation than in the nonplan-like situation if the assumption of how experts build their programming schema was correct. Soloway and Ehrlich collected data from 139 students: 94 novices and 45 advanced programmers, and concluded that the result did support their assumptions.

Rist (1989) extended the concept of programming plan to the creation of programming schema by analyzing verbal protocol and code protocol from 10 novice programmers’ eight Pascal programs. He proposed a cognitive structure of programs which had four levels: (1) from code fragment to single line, (2) from line of code to programming plan, (3) from programming plan to program, and (4) abstract program structure. The lowest level of knowledge construct is fragment of code which can be combined to form a line of code. Lines of code can then be combined to form programming plans. Each programming plan has an associated focal line. When a program is developed, experts tend to develop focal lines before non-focal lines. Using this model, a program can be represented by an abstraction structure and that simple knowledge construct can be combined into a more complex construct. For novice programmers, the development of programming knowledge is very likely starting at the lowest level because novice programmers have only fragmentary knowledge and no programming plans yet. On the other hand, experts, because of their bountiful programming plan knowledge, usually develop programming knowledge from generating programming plans.
Although the programming plan was proposed as a major feature of programming expertise (Rist, 1986; Spohrer, Soloway, & Pope, 1985), Gilmore and Green (1988) argued that programming plans are a knowledge construct that is language-dependent because previous studies all used Pascal, or similar types of programming language experts. To test whether the programming plan was a construct underpinning programming knowledge across different programming languages, they used 32 experienced undergraduate programmers some of whom knew Pascal and some who knew BASIC and gave them different perceptual cues—indentations and color. Participants’ performance was measured by their error detection rate. What they found was that these perceptual cues were useful to experienced Pascal programmers, which suggested that programming plan was part of expert Pascal programmers’ programming knowledge and could be a unit of programming schema. Perceptual cues, however, did not improve expert BASIC programmers’ performance statistically. Gilmore and Green (1988) argued that a property of programming language called “role-expressiveness” determined whether expert programmers constructed programming plans. Experts of Pascal have programming plan expertise because Pascal is a role-expressive language. On the other hand, BASIC is not a role-expressive language. Therefore, they argued, experts in BASIC language do not form programming plan in their schemas. In essence, the property of role-expressiveness enables programmers to develop automatic skills of mapping between the content knowledge and the problem to be solved. The findings suggested that the occurrence of programming plans may vary among different types of programming languages. Thus, the programming plan is not a universal programming knowledge construct.

Davies (1990) questioned both the view of plans as universal cognitive representations of programming and the view of plans as the result of certain programming language characteristics. He proposed that the learning experience of programming design knowledge was closely related to the acquisition of programming plans rather than the notation of programming language.
Because different groups of learners learn programming languages with different emphasis, the learning experience varies. For example, when computer science students learn programming, functional decomposition and other design skills are usually demonstrated by the instructors. However, students from other majors are usually taught only surface-level programming knowledge, according to Davies (1990). There is also a difference in what programming languages are taught in computer science versus other majors. Gilmore and Green (1988) ignored the learning process and experience, but drew conclusions based on the notation of programming languages. Davies (1990) proposed that instead of accepting programming plans as either natural artifacts or notational artifacts, the acquisition of design skills was what influenced how expert programmers acquired and used programming plans. He conducted two experimental studies to test his hypotheses. In one study, he divided 72 expert programmers equally into two different groups: one group of students (group A) were computer science major undergraduate course students and the other group of students (group B) were students from accounting, finance, and engineering undergraduate courses. Both groups had a similar level of BASIC programming experience except that participants from group B had no experience in program design and participants from group A had attended a program design course. All participants were given BASIC programs with different types of structural cues—no cue at all, cue on control structure, and cue on plan—with different types of bugs: surface, control structure, and plan. When measuring the percentage of error detection and correction rates, Davies found that “the results clearly demonstrate the effect of the possession of design-related skills on the comprehension of plan-based structures in programs” (1990, p. 473).

In the second study, Davies used two groups of 24 first year undergraduate students who had at least six months of BASIC programming experience before the study. One group attended a program design course and the other did not. Each participant had two trials to identify and to correct critical lines in a 30 line BASIC program. The first trial was done before a programming
design course, and the second trial, five months after the first trial, was done after the course. Their performance indicated, as Davies expected, that the only interaction between programming plan expertise and plan-like programs was when participants attended the programming design course. For the group that did not attend the course, there was no effect on participants’ performance between plan-like and nonplan-like programs.

To summarize, expert programmers have a better strategy in knowledge organization than novices have. Their strategy of organizing information gives them an edge on storing and retrieving knowledge more efficiently. Assuming both experts and novices have the same amount of STM capacity and information being retrieved from LTM into STM, this chunking strategy can allow experts to access more relevant information, whereas novices have more irrelevant information which limit novices’ problem solving ability. In general, there are two propositions that can be observed from previous work: (1) the unit of chunking is through semantics grouping not symbolic binding (i.e., function and not syntax), and (2) the strategy of chunking can be acquired through learning regardless of the properties of the programming language.

**Strategy**

Programming schema theory is primarily concerned with programming knowledge representation, content, and structure. The result of those studies—programming plan, programming schema, or program template theories—describes the characteristics of the declarative knowledge structure, but often ignores the different strategies that programmers used. The limitation of these knowledge-based theories, therefore, is that they discount the role and the development of different programming strategies of programmers (Davies, 1993). There are other components underpinning programming schema theory that also control how and what programming knowledge is used. One of the assumptions of programming schema theory is that
the development of programming expertise is the acquisition of programming plans and the
cognitive process of programming is somewhat unified. From the perspective of programming
schema theory, the development of programming expertise is the acquisition and structuring of
programming plans. The difference between experts and novices, from the perspective of
programming schema researchers, is the number of programming plans possessed. However,
McCracken (2001) reports that many students learned programming knowledge and concepts
well, but still cannot write a good program. There must be other components that interact with
programming knowledge and influence the application of that knowledge. After all, software
development ability should be more than what programming schema can describe and predict if
software design is distinct from programming alone.

Software design, like other design activities, is a process of solving ill-defined problems.
Researchers are often trying to identify a pattern, hopefully an optimal one, of software design so
that it can be used to train software designers. Researchers in the area of the psychology of
software development often borrow the concepts from generic human problem solving theories,
such as a production system that can simulate and predict human behavior during a problem-
solving situation (Brooks, 1977; Cohen, Ritter, & Haynes, 2005; Jeffries et al., 1981). To
construct a problem space from a complex problem, the solver usually divides the problem
statement into sub-problems. These sub-problems consequently form a smaller problem space to
be resolved through manipulating actions, conditions, and operations. When the sub-problems are
solved, the problem solver has reached the designer’s goal. How does a software designer form a
problem space? That is, what strategies does a designer employ to divide a problem statement
into sub-problems? One commonplace view is that designers use a systematic, top-down global
decomposition. Visser and Hoc stated, “a top-down strategy consists in descending the solution
trees from the most abstract level down to the lowest, concrete level” (1990, p. 241). Wirth
(1971) suggested a “stepwise refinement” approach in teaching program development and
provided an 8-Queens problem to demonstrate the stepwise processes and to generalize and extend the processes into other problems. The concept of the stepwise approach or the top-down strategy or structured programming (Dahl, Dijkstra, & Hoare, 1972), became a movement in rigorous program design from the 1970s onward.

In addition, researchers were able to observe the experts’ use of top-down decomposition strategy in various empirical studies (Adelson, 1981; Brooks, 1977; Jeffries et al., 1981), but inconclusively. For example, in the study by Jeffries et al., there were two out of four participating expert programmers that had deviated from top-down decomposition strategy in their design activities.

The behavior of some experts’ deviation from top-down decomposition strategy made some researchers (Visser & Hoc, 1990; Visser, 1990; Visser, Olson, Sheppard, & Soloway, 1987) believed that expert programmers might be more “opportunistic” rather than strictly top-down. Motivated by a lack of real world “programming-in-the-large” studies, Visser (1987) observed an expert programmer (a mechanical engineer) full-time for four weeks while the programmer did work on controlling an automatic machine tool installation, particularly in the programming strategies for defining functional specifications. The programmer was asked to “think aloud” as much as possible during the process of design and programming, and the think aloud video became the primary data source. As Visser reported, even though the programmer said he followed a structured plan, this programmer did not demonstrate systematic top-down design behavior in situations such as new information discovery and cognitive difficulty. The design activity was, instead, opportunistically organized. Guindon (1990) also analyzed three expert programmers’ verbal protocols of designing the logic of a Lift Control Problem. The expert programmers approached a high-level design solution by writing down notes, diagrams, descriptions, etc.; coding was not necessary. To record the progress of their design behavior, the
design documents were regularly time stamped. Similar to what Visser (1987) had found, Guindon concluded the reason for opportunistic design behavior:

The main causes for opportunistic solution development include immediate recognition of a partial solution in another part of the problem, immediate handling of inferred or added requirements, drifting through partial solutions, and interleaving of problem specification with solution development. (1990, p. 327)

The reason that experts use opportunistic rather than top-down strategies is because of the ill-defined nature of design problems. Guindon (1990) suggested the reason other researchers were able to observe structured top-down decomposition behavior was due to the simplicity of the design tasks. The artificial and relatively small design problems being used in some studies (Adelson & Soloway, 1985; Brooks, 1977; Jeffries et al., 1981) provided a structure that made top-down design strategy possible and a reasonable choice for experts. But when experts faced ill-structured design problems, they chose to move away from top-down strategy as they “inferred new requirements and discovered new partial solutions” (Guindon, 1990, p. 337), which were not readily available due to the problem solutions exceeding experts’ STM capacity. The failure to retrieve the needed information from LTM also influenced the ability of expert’s mental simulation and sometimes made the programmer seek the best opportunity to solve the problem. Therefore experts who designed an unfamiliar object in an unfamiliar domain were more likely to demonstrate the opportunistic strategy (Adelson & Soloway, 1985) because there was no existing solution in experts’ programming schema. Some researchers (Hayes-Roth, Hayes-Roth, Rosenschen, & Cammarata, 1979; Hayes-Roth & Hayes-Roth, 1979) suggested that the type of problem might influence the selection of different strategies by the designers of different levels of expertise. However there is no empirical data to support or reject this conjuncture.

Different conclusions from earlier observations about programming strategy suggest two strategies, either top-down or opportunistic. It is inconclusive to determine what programming strategies are being employed globally by what skill level of designers. In response to the debate
of top-down or opportunistic programming strategy, Davies (1991b) conducted a quantitative study on two groups of programmers, one group consisted of 20 novice programmers and the other consisted of 20 expert programmers. Participants were asked to solve different programming problems of varying difficulties. He defined non-linearity by jumps between focal lines and jumps between non-focal lines (Rist, 1989). The non-linear jumps violated the hierarchical structure of the top-down model of design and, therefore, should not occur often if programmers employed top-down design strategy. On the other hand, jumps from focal line to non-focal line or vice versa were considered to be switching between different abstractive levels of hierarchical structure and, therefore, would predict top-down design behavior. He recorded participants’ frequency of switching between focal and non-focal line in code generation during different generation blocks (time) and concluded that both programming strategies were used in different stages of code generation. For example, expert programmers adopted a top-down strategy in the early stage, but then demonstrated more opportunistic strategy later in the design process. This design behavior was in accord with Guindon’s observation (1990) of the origin of opportunistic strategy that experts acquired new solutions or inferred new requirements during the design process. Davis (1991b) elaborated on the concept of design strategy and proposed that design process was neither a top-down nor purely opportunistic for either experts or novices. Designers started with a more top-down model of design and later sought better opportunities to solve the problem. We do not know when or how designers switch from one strategy to another, but experts demonstrate opportunistic strategy more often than novices do.

So, in the end, there is no definite and direct empirical data yet on what strategies are employed at what level of expertise. However, the opportunistic design strategy is related to new information, new requirements, and cognitive difficulties. Therefore, we can expect the increase of opportunistic design behavior if: (1) a problem is ill-structured, (2) a designer possesses more design-related knowledge, and (3) a combination of both (1) and (2). In fact these three
conditions can converge into condition (1) because, according to Simon (1973), the definitive factor of what constitutes an ill-structured problem is the amount of knowledge the solver has.

**Mental Models**

Another important component in the psychology of software development is a human’s ability to make an internal representation of the external environment. Conant and Ashby (1970) established the necessity of model-making by the human brain in a complex dynamic system though set theory. They concluded that “a regulator must model what it regulates” and we should now focus on its efficiency. Although Conant and Ashby’s study was not directed at software development or problem solving, they were confident that their conclusion could be applied to any complex dynamic system. Software development, without a doubt, is a complex task, and it is also a dynamic system. Software development tasks are usually ill-structured, ill-defined, and have undetermined solutions in a large problem space (Guindon, 1990). These uncertainties will shift as new information and knowledge change the problem space (Simon, 1973). As a result, the path to the solutions is by nature a dynamic one. In other words, when facing a complex problem like a software design situation, designers must have a mental model (Johnson-Laird, 1980) of what the system is and how its components interact with each other. A mental model enables a designer to act on different situations and to select the best solution by simulating the outcomes of potential actions before they are selected. In a series of three empirical studies by Kieras and Bovair (1984), they randomly assigned 40 participants who they recruited from campus into two groups evenly. One group (model group) studied model materials for a device control panel until they passed a knowledge test. The other group (rote group) did not. Both groups then received operating procedures on operating the device control panel. They found that the model group performed the tasks more efficiently and were able to optimize some of the procedures that were
made inefficient deliberately. In a following study, the participants were asked to think out loud while inferring procedures. The participants from the model group had better reasoning ability compared to those who were from the rote group, and the model group outperformed the rote group by the number of successful attempts. In their third study, they concluded that it was the system topology that helped the model group out-perform the rote group. To summarize their studies, they concluded that mental models had a positive influence because the task became meaningful. Mental models helped the solvers understand how the system worked.

In general, a mental model represents a person’s internal representation of a situation, surroundings, or a domain. In a problem solving situation, a mental model will help the solver in comprehending the problem, inferring hidden features, and optimize the operations (Jeffries et al., 1981). However, a person’s mental model can consist of only partial information and may be subject to misunderstanding (Gentner, 2002; Kieras & Bovair, 1984). That is, everyone, more or less, has a mental model for different situations regardless of how accurately the mental model reflects the reality. A mental model may be incomplete because it is sufficient to the owner and there is no reason to change it. For example a novice programmer may have a mental model about the computer in which it takes a command and executes it. But for experts, the mental model of the computer can be at a more detailed level—such as how memory communicates with the CPU and how different types of variables are handled differently by the compiler—because their mental models are reinforced by prior experience. The novice’s mental model does represent part of the reality, but in a much simpler way. The accuracy of a mental model helps experts excel in a specific domain. A mental model can also be called incorrect while it violates the important principles of reality. For example, from Piaget’s experiment on children, some younger children had the mental model that it takes more objects to form a longer line, and they ignored the spacing between objects. This mental model is incorrect before the spacing is included, and it will mislead the children’s decision-making process.
With that caveat, people with different levels of experience will construct distinct mental models subconsciously, which is a representation of reality, because there are no globally accepted mental models for every subject. To regulate and prevent the creation of incorrect mental models, the formation of a mental model should be part of the discussion in the learning process, especially a process like software design, which involves a lot of mental work. After all, the mental model is a special type of knowledge, the one that describes a particular system with which a person interacts. It helps a person understand, control, and estimate the state of the system being acted upon. Rouse and Morris (1985) referred to mental models as a function of describing, explaining, and predicting the status of a system. Learning a programming language and constructing a mental model of how that language will be executed by the computer often occur simultaneously (Hoc & Nguyen-Xuan, 1990) except that mental models are usually constructed subconsciously. Design-oriented mental models are highly situational and constrained by experience and existing programming schema. In many situations, it is a sufficient explanation of human behavior (Rouse & Morris, 1985).

Research can only, at best, estimate mental models from various sources. Empirical study can be one method to infer the structure and characteristics of mental models by designed activities and the participants’ observed behavior or measured performance (Fix, Wiedenbeck, & Scholtz, 1993). Many tasks, however, are not suitable for empirical studies to infer the input-output relationship. These tasks require a lot of cognitive processes, and their output is not exclusively determined by the mental model involved in the process. Tasks like problem solving and decision making are in this category. Situations like these call for ‘direct inquiry’ methods (Rouse & Morris, 1985). In direct inquiry, researchers ask participants what they are thinking about concurrent to performing a task. Verbal protocol is recorded and used as the primary data to understand their mental models. While self-report data can also have problems, real-time think-
aloud protocols during a problem solving task are often the simplest way to access certain internal cognitive processes or mental models.

**Summary**

In this chapter, I introduced basic human cognitive architecture, problem solving, and production system theory that uses rules to represent knowledge. Three psychological aspects related to software development followed. In the knowledge section, I discussed programming plans as a possible construct of programming knowledge and the generation of such plans. The conditions of plan acquisition were also discussed. In the strategy section, two views that are common in programming behavior were discussed. Some researchers advocate structured programming or top-down design, others believe more programmers use more or less opportunistic strategies. Although there was no definite conclusion, the structure of a problem and the knowledge of a solver both dictate which strategy would be overt. Finally, I discussed the existence, the function, and importance of a mental model for a designer, or for any problem solver.
Chapter 3

METHODODOLOGY

“Before we can test any theories, we need to build some. Before we can build some theories, we need to better understand the inspection process as it exists in the real world.” (Letovsky, Pinto, Lampert, & Soloway, 1987, p. 232)

This study explores the formation of mental models involved in a software design task done by novice programmers, and the effect of those models on the design task. This is done by examining the cognitive processes involved. There are two main methods of studying mental models. One is to ask participants about their beliefs through interviews and questionnaires, and the other is to analyze think-aloud protocols (Gentner, 2002). I employed the latter because software development tasks are knowledge-rich tasks, and they might be difficult to articulate retrospectively, especially for novice programmers. The validity and reliability issues of using verbal protocol analysis (VPA) are discussed in the articles by Ericsson and Simon (1980, 1984). Other studies in the same vein use the VPA as well (Brooks, 1977; Chi, Bassok, Lewis, Reimann, & Glaser, 1989; Guindon, 1990; Jeffries et al., 1981).

Although novices are the primary target of this endeavor, experts’ data is collected for two reasons. First, it serves as baseline data for novices’ cognitive capability interpretation. For example, if there is a discrepancy between the novice and the expert in their cognitive capability, then the discrepancy might be one of the reasons for experts being expert and novices being novice. On the other hand, if there is no observable discrepancy, there must be some other factor contributing to the difference of expertise. Secondly, experts’ verbal protocol can inform us in selecting coding themes. Examining design expertise will provide a wider range of cognitive activity, which can blanket what will be found in novices’ cognitive capability. Therefore, experts’ verbal protocols are also collected and analyzed.
Participants

There are two types of participants in this study: expert designers and novice designers (expert and novice hereafter). Novices are designers who are in the early stages of learning computer programming and who are pursuing information technology or related areas as profession. There were about 45 undergraduate students in each of the two sections of IST240 Introduction to Computer Languages at The Pennsylvania State University main campus. IST240 is a required course for all Information Sciences and Technology (IST) majored students and is designed for the second year of the IST major. The prerequisite course for IST240 is CMPSC101, where most students learn C++ for the first time. Therefore IST240 is the second course that focuses on computer programming for many IST students before they take on more advanced software design courses. In IST240, students learn the Java programming language, as well as design tools such as Unified Modeling Language (UML). In-class labs and final projects are where students gain their hands-on programming experience. Students from both sections were recruited, and in the end, there were eight students who signed up for this study. The experiment time for students was 3.5 to 4 hours in total, and 100 dollars, prorated for each data collection session, was provided as an incentive. All eight student participants remained in the study until the end of the study. Their individual demographic data will be discussed in detail in chapter four. Two instructors from IST240 were recruited as experts. Their qualifications and demographic data will also be discussed in chapter four.

In terms of cognitive activity, novices are expected to have some changes and progress, and this progress should contribute toward building their domain expertise. Both the lack of programming knowledge and cognitive limitation types of breakdowns that experts experienced (Guindon, Krasner, & Curtis, 1987) are expected in novices during their design as well.
It is a common practice that studies in software design use one or more design scenarios as problem solving tasks, superficial or realistic, and then the thinking processes are recorded and analyzed based on different research questions. Some of the design scenarios could be solved with less than one hundred lines of code, and were criticized for being artificial and unrealistic, because those problems might not require complex cognitive activity (Visser & Hoc, 1990; Guindon, 1990). To avoid some of those pitfalls, the major instrument for this study is a design scenario that satisfies the following criteria:

- **Complex**: To elicit different cognitive processes, the problem needs to be complex. If the problem only involves a simple solution such as counting a running total, then the only observable cognitive process is very likely to be applying a canned solution.

- **Domain knowledge independent**: By domain knowledge independent I mean the solution does not lean heavily on mathematical calculation, understanding some complex physical phenomenon, or using some specialized algorithm. When experts engage in problem solving tasks in an unfamiliar domain, their performance is no better than novices as observed by Soloway and Ehrlich (1984). Although domain knowledge is an important factor in a design process, it should not be the only one or dominant one.

Participants received a design scenario, a modified design task from Adelson and Soloway’s study (1985), which required the designer to construct a blue-print of an online library management system (OLMS). An OLMS is similar to a typical library management system, which allows patrons to search for library items, lookup their own library records, change personal information, reserve library items, and renew checked out items. It also allows librarians
to check out library items for patrons, send special notifications to patrons, and modify library items, in addition to all the functions available to patrons. The goal is to create a blue-print for such a system so that the design document can be handed to programmers for implementation. Designers, therefore, only need to create a proper design document for implementation, and they do not need to write computer programs. The detailed design scenario and instructions are listed in Appendix B.

Let turn our focus to how the design scenario fit into the criteria described previously. First of all, to complete this OLMS, different real-world entities, such as patron, library, and library items, need to interact together. Different things are involved in different situations to accomplish different functionalities and, in turn, achieve the overarching goal. Each of these real world entities will be reflected somewhere in the computer software, and in most modern software environments would be represented as objects in the software. There are about three different objects with about 13 functions (depending on whether some of them are reused or not) involved in the OLMS. It is so complex that a canned solution or a template will not solve the problem. Although some objects or functions may be familiar to the designer, it is unlikely that she has this whole system in her memory. This scenario, therefore, is complex enough bring out different cognitive processes from the solver. Problems that are complex enough to not be memorized by human’s limited cognitive load are a better choice for the purpose of this study.

Secondly, a library system contains at least a set of common concepts, such as borrowing and returning, even for people who have never used a library service. For college students, it is a fair assumption that they understand how a library works. Therefore, OLMS should be a familiar task for the participants so that it does not tax extra domain knowledge nor add unintentional cognitive complexity to the task. There will be a significant difference among individuals if the task is a familiar one to some, but not all of the participants. For participants who are not familiar with the problem to be solved, it will be difficult to apply their design knowledge, and their
design processes may be unobservable. Although in reality, software design tasks are usually across domains, it is not the intention of this study to investigate the interaction of cognitive processes of design task and domain knowledge. The OLMS will allow designers to focus primarily on the design aspect of the task without worrying about understanding specialized domains or algorithms.

In addition to the design scenario, a demographic survey was also used to document every participant’s demographic data, especially the use of computers in daily life, and prior design and development experiences. The survey data can help us understand whether participants transfer knowledge from other experiences and how their prior knowledge affects the assimilation and accommodation process (Rumelhart & Norman, 1978). Questions are designed to understand what computer applications the participants use in general, what programming experience they have, and what programming language they have used before. For example, participants who use spreadsheet formulas in Microsoft Excel often may possess superior problem decomposition skills and, hence, transfer to software design. Appendix B lists the protocol of the survey questions. The survey interview is conducted face to face, and some additional questions are asked to gain deeper understanding of their computing background based on participants’ responses.

**Procedures**

All participants participated in three study sessions during one semester and each study session included a software design session. Originally, a video review session was planned after the design sessions, but it was dropped in the third session because all but one (David) of the novices did talk aloud well after the second design session. In David’s case, the lack of talk aloud data was because of his lack of domain knowledge. An additional review session would not help
after all. In addition, any retrospective reports that were not given immediately after the task were less reliable because the specific information was not available in the STM anymore (Ericsson & Simon, 1984).

All three study sessions follow the same procedure except the first study session additionally included a demographic survey (listed in Appendix B). I explained the study to the participants and obtained their written consents in the beginning of the first study session, which was held in the second week of the fall semester. The demographic surveys were audio recorded for transcription later. After the survey, I explained “talk aloud” and showed a four minute video that contained both a good talk aloud and a mediocre one to the participants. The participants were given the online library management system design scenario as the design task and a pad of letter size paper and two pencils to use to describe (draw a diagram or write down the description) the design of the system during the think aloud explanation. The participants were instructed to stop only when they could not improve the design anymore; participants were also told they were free to discard the design at any time. The discarded papers were collected and numbered for analysis later. Video recording was used during the design process with the camera focusing on the paper that the participant used in case the participant failed to think aloud. I asked the participant questions such as: Why do you do this? How do you know you need that? What makes you do this? These non-leading questions were used to force the participant to describe their design cognition verbally.

After the first and the second software design session videos were transcribed, the novices and I reviewed the video together to discuss the times when the novices were not talking out loud during the software design session in order to get further explanation. The review sessions were dropped after the first two sessions because of the reasons mentioned earlier.
The second study session started in the seventh or eighth week of the semester when the IST240 class was at about the half way point. In the beginning of this design session (in the third design session as well), the novices were given the following additional instructions:

(1) Assume a database that supports this online library management system already exists.
(2) Your task is to design the system between the database and the user interfaces.
(3) Forget about what you did last time, start the design from scratch.

Statement (1) and (2) were added because some novices seemed to carry their database experience from IST210 class into this design. Statement (3) was to remind the novices to start a new design so that previous design would have less interference with the current process. During the second study session, all participants designed the online library system again under the same conditions as in the first study session. The third session was held in the thirteenth week of the semester, one week before the class was over, using the same conditions and process as previous study sessions with the participants working on the same online library system design scenario.

The participants were presented the same design scenario three times during the study and it was very likely that some of their improvement is because of the learning effect from repeating the same problem, rather than the course or other learning experiences. The purpose of this study is, however, to explore the progress of novices’ mental models, not the source of the learning or to isolate the learning and factors that cause it; hence, it is appropriate to collapse their improvement because of repeating the same task with improvement from other causes.
Pilot Study

Participants

A pilot study was conducted the semester before the main study to gauge the effectiveness of the design scenario and the time it takes for one study session. Three students, all males, from COMPS 201: Programming for Engineers with C++ at Pennsylvania State University participated in the pilot study. They were offered 20 dollars for their participation in the pilot study. All participants stopped their design within one hour when they felt they could not improve the design anymore or they believed the design was good enough to be used as a blueprint for programmers.

Pseudonyms, Bell, Ian, and Andy, are used to represent these three participants; they were novice, intermediate, and advanced programmers respectively. Bell was a new international graduate student arriving at the university that semester, majoring in engineering but with no programming experience at all. CMPSC201 was the first programming course he had ever taken. The only programming experience, indicated by Bell, was using Microsoft Excel to generate data. Bell, therefore, is considered to be a novice who does not have any formal programming training.

Ian was a returning student who was going to be attending graduate school the semester after the pilot. He had a course that required him to use TrueBasic, a programming language similar to BASIC, for his course project. Many of his friends were technologically savvy, so he gained knowledge from conversing with them. Ian’s programming competency was viewed as similar to students who would have just entered the IST 240 course.

Andy was a junior undergraduate student majoring in Computer Science. He has written many computer programs since high school. He was very comfortable developing programs individually and is very confident in his software development ability. One of his previous
projects was a computer game written in C++ using the DirectX library. He spent his spare time online reading technical reports, involving himself in forum discussions about programming, and developing small applications by himself. He was not enrolled in CMPSC201, but received the information about this pilot study through a friend.

**Procedures**

In the beginning of the study session, the participants were asked a set of questions about their computing background and programming experiences based on the interview protocol. As a result of their responses, additional questions were asked to understand their programming background better. Their responses were audio recorded and were used to explain the software design behavior and strategies.

Before the design task started, I explained think-aloud to the participants before every study session. The explanation stressed the importance of talking aloud not only on what they do but also the reasons behind their actions. I also reminded them that in the case of the participants failing to think aloud, I would ask questions to uncover the reasons behind their behavior.

The design task using the online library system scenario followed the think aloud explanation immediately. All participants studied the scenario before starting their design. A pad of paper and a pencil were available to the participants to record their design. They were instructed to use any kind of method, such as writing text descriptions or using diagrams, as long as they could construct a design document that could be used as a blueprint to implement the system by programmers. There was no limitation on how many pages a participant could use. At the end of the study session, the design papers were collected and numbered.
After the design scenario was over, the participants were asked for their feedback about the design scenario with regard to difficulty and clarity. This part of their response was to improve the quality of the design scenario to be used in the upcoming study.

**Findings**

All three participants demonstrated different design strategies. The decomposition from the beginner covered only surface features of the system. The intermediate programmer attempted to decompose based on the system’s functions and tasks, but could not go further because of his limited programming knowledge. The expert showed a lot of design skills by not only systematically decomposing the problem but also carefully reconstructing the solution to improve it. All three participants finished their design session after between 45 minutes to one hour. The differences were that the novice seemed to focus on many surface features, while, on the other hand, the expert spent almost all of his time on organizing the underlying structure. From their verbal protocol, I believe their mental models of the problem were limited by their programming knowledge and experiences.

The main purpose of the pilot study was to evaluate the appropriateness of the design scenario. Again, all participants spent between 45 minutes to one hour in the design activity. Had it been too difficult, Bell would probably spend much less time because he could not move forward. On the other hand, had it been too easy, Ian would spend much less time because he should be capable of finishing it early. So, in terms of difficulty, it seemed appropriate for the participants at all skill levels. Furthermore, because it took Bell, Ian, and Andy about the same amount of time, their programming skill levels were independent of the design scenario. As a result, the task was considered to be domain independent enough for the main study. From this point, we will return to the discussion of the design of the main study.
Data Analysis

The primary data collected and analyzed in this study were video recordings of designing an online library management system. Participants’ design artifacts were used to assist the analysis as well. Although the participants were being video-taped, the video camera only focused on their design document, so the facial expression and gestures were not analyzed. Therefore, the analysis is a verbal analysis rather than video analysis. The unit of analysis is to the smallest task of same context the participants are attending to during the design process. Their estimated cognitive processes were distilled from the verbal protocols and described with respect to the change and evolution of those estimated cognitive processes.

Several methods can be used to analyze verbal data. One way is to use verbal protocol analysis (VPA) (Ericsson & Simon, 1980, 1984), which has been used in analyzing think-aloud data similar to this study. VPA is aimed at discovering an optimal path of problem solving in which sequencing is paramount. The sequences or paths to a solution can usually be enumerated before human data is collected because a problem space is normally bounded in a production system. The main purpose of VPA is, then, to inform cognitive model creators as to how a model should attend to a problem to find the solution. The creators can tweak parameters, in the case of their model, so as to select a different path in order to make their model match what humans, usually experts, will perform. In this study, however, the purpose is not to discover the path of how a participant, either expert or novice, will approach the design scenario. What is important to understand in these cognitive processes is the participants’ knowledge representation or mental model, which supports their design activities related to the online OLMS. VPA was, therefore, not selected to analyze the verbal data collected in this study.
Another analysis methodology that is in a similar vein to VPA, coined as verbal analysis by Chi (1997), has a different emphasis and is more appropriate for this study. This type of qualitative data analysis method is used to “capture the representation of the knowledge that a solver has” (Chi, 1997, p. 277). Adapting from her proposed guideline, the following functional steps were taken to analyze the data after all data was transcribed:

1. Segmenting protocols by episodes, tasks, and contents.
2. Developing a coding scheme or formalism.
3. Identifying evidence in the coded protocols that constitutes a mapping to some chosen formalism.
4. Depicting the mapped formalism for all sessions across all novice programmers.
5. Seeking pattern(s) in the mapped formalism.
6. Interpreting the pattern(s).

All video and audio recordings were first transcribed into textual protocols. Each protocol was then segmented into episodes based on the context and tasks to which participants were attending. The reason to segment protocol by context rather than proposition is to preserve information that is related to the same task. Each episode was then categorized into one of two separate tasks: one is related to the functions of the system and the other is related to the non-functional part of the system. In the functional design task, there are five major categories: problem comprehension, decomposition, structuring, mental simulation, and evaluation. More detailed information for each category is discussed in the section on coding themes later in this chapter.

Every episode was labeled by a single rater, according to the coding scheme; this reviewer read the content and classified the participants’ cognitive activity. All adjacent episodes with the same label were reviewed again to ensure they belong to different episodes. Long
episodes also received another inspection to confirm that it belongs to one episode, rather than two episodes.

After all the episodes were labeled, a graph was generated based on the labels to be used for identifying patterns. Individual cognitive processes were then interpreted through all three sessions of the software design episodes. In addition, experts’ patterns were provided as a reference to the novices’ cognitive patterns. However, the process was not a linear process. Additional themes were discovered and added to the coding theme. Segmentation was also cyclical by further segmenting episodes into smaller episodes when appropriate.

Non-design is attributed to the time when the participants are not attending to activities that are directly related to solving the design scenario. Non-design activities are verbal data related to general design principles (e.g., size of buttons and layouts about interfaces), tasks the designers would like to make available, but is not in the instruction (e.g., make something similar to Microsoft Surface), and conversations between the participants and the experimenter that are unrelated to the current design task. Although those situations are related to software design, it is not directly related to the current design problem per se. The non-design episodes were analyzed qualitatively and discussed separately from other episodes.

**Coding themes**

The following cognitive processes emerge from previous literature (Brooks, 1977; Guindon et al., 1987; Jeffries et al., 1981; Letovsky et al., 1987) and from analyzing experts’ verbal protocols. The process of analyzing verbal protocol is cyclical between the analyses of verbal data and the generation of coding themes until no more themes emerges. These coding themes are used to establish the categorization of each individual’s verbal protocol episodes.
Problem comprehension

Problem comprehension is the process of translating a set of requirements (problem description) into an organized internal representation (mental model), through existing schemas, prior experience, intuition, and other possible ways. According to schema acquisition, encoded knowledge stored in LTM is an individualized mental representation of a design scenario. Problem comprehension is the cognitive process of bridging the external world and the internal representation. A problem can only be understood by accommodating or assimilating to existing schemas (Rumelhart & Norman, 1978). Because humans are selective in SM (Driscoll, 2000), what information is filtered into STM is critical to the formation of a mental model. Therefore, problem comprehension is the first cognitive process in the software design process.

The mental model at this stage is in a preliminary format, incomplete or incorrect, and will be modified and restructured during the design process. In this study, it may be difficult to form a complete mental model because there are two cognitive tasks competing with each other for attention: text comprehension and mental model generation. Because the text comprehension is the primary task, we can expect mental model generation to lag behind because of the high cognitive load in the comprehension process (Sweller, 1988). As a result, a change of mental model and a repeat of text comprehension are both expected later in the design process.

Decomposition

Decomposition is the process of generating intermediate subsystems, subcomponents or procedures which together can achieve, at least a portion of, the final goal of the design. There are different types of decomposition. A designer can divide a problem into smaller problems so that they are more manageable. We will call this problem decomposition. It is a common strategy
and can be observed in a problem solving process whether it is means-ends or difference-reduction (Anderson, 1985). For example, a library system can be divided into tasks that fulfill the needs of the librarians and the needs of the patrons. Each need can be broken down further, depending on the designer. The designer can also divide a function into smaller steps, and we will call this *procedural decomposition*. Depending on the design, she can see a task as a complete piece or use procedural decomposition to examine its internal mechanism for reuse or for restructuring. For example, the function of checking out a library item can be broken down into: checking the availability, verifying a patron’s identify, calculating a due date, modifying the library or the patron’s personal records, etc. In general, there will be new information being added into the current mental model during the process of decomposition. The mental model is expected to be examined in detail and to expand further.

**Structuring**

Structuring is the process of examining and associating the relationships and interconnections of subsystems, subcomponents, and procedures that are created from decomposition. Unlike decomposition which concentrates on one component and breaks it down into several subcomponents, structuring concentrates on connecting existing components. It is often represented by arrows pointing from one component to another. Collaboration is the main goal in this cognitive process. As a result, structuring requires the designer to heed information from several components and to manipulate their properties in a way that will achieve one task. A higher cognitive load may be required in this process because multiple components have to be brought into working memory at the same time.
**Mental Simulation**

Mental simulation is a process that uses a hypothetical input and executes it in part of the design to see if the output is as expected. It is often used as, but is not limited to, a method of verifying the product of structuring. Software designers use two types of mental simulation which requires different kinds of knowledge and leads to different design behavior (Letovsky et al., 1987). The first one is *scenario simulation* where designers anticipate the system output based on user input. A designer can imagine herself as being a patron using a library system. She will need a working mental model in her memory to “see” and “operate” the system. Another type of simulation is *functional simulation* where designers execute a partial or complete system to insure the sequences of a functional procedure are coherent.

**Evaluation**

Evaluation is the process of making a judgment about selecting alternative solutions. Typically an evaluation process includes phases such as identifying the problem, generating a hypothesis, testing alternatives, and choosing a solution. However, these phases are not always included in an evaluation process. The designer may decide to postpone an issue until she has better understanding. There can be several benchmarks which include efficiency, consistency, redundancy, etc, in an evaluation process. Evaluation can also be based on intuition because we believe that some designers may have sufficient knowledge that automates the evaluation process. In such cases, the evaluation process can be implicit and seems intuitive to the designers.

These five codes are used to categorize cognitive process episodes segmented from the verbal protocol. I expect that problem comprehension, decomposition, and structuring activities will compose the majority of the total cognitive activities because they are common in problem
solving situations. Mental simulation and evaluation, on the other hand, require more complex cognitive ability and designers can finish a draft design without including these activities.

Activities not related to solving the problem directly, such as describing the size of a button or using which technology to use to create the system are all categorized as non-design activity.

Reliability

In the world of qualitative data analysis, no one can escape from the criticism of subjectivity. This research study is no exception. Nevertheless, I shall strive to limit the chance of such a negative impact in making meaning out of non statistical data. Subjectivity can arise from rigorous theme coding when statements are made in an ambiguous context. Having a systematic process to code such a situation will improve the coding result and improve consistency. Some statements may seem to fit into several themes depending on perspective. Table 3.1 provides examples, corresponding themes, and the rationales. This table is used when there is doubt about certain episode.

Table 3.1
Ambiguity Examples and Solutions

<table>
<thead>
<tr>
<th>Example Episode</th>
<th>Coded Theme</th>
<th>Rationale</th>
</tr>
</thead>
<tbody>
<tr>
<td>I have completed reserving books.</td>
<td>Problem Comprehension</td>
<td>Comparison of represented system and representing system.</td>
</tr>
<tr>
<td>Patron brings a book, types book ID, the system looks up records, system returns book info…</td>
<td>Decomposition</td>
<td>This is a typical functional decomposition although it may look like a structuring or mental simulation.</td>
</tr>
<tr>
<td>I will put a big RESET button here so everything can be back to where</td>
<td>Non-design</td>
<td>This is not directly related to the structure of the solution.</td>
</tr>
</tbody>
</table>
you saved it.

| Since I guess the way my library at home worked, if you got a book, you have it for two weeks, but if you had like a video or a CD, it’s only one week. | Problem comprehension | This is an example of using existing knowledge from long-term memory to construct a representing system. |
| So overdue notice you authorize this library only with access as both the customer who is limited to self only because it’s sensitive information. And the library has to . . . that way they could change it if necessary and the customer cannot change it. Again from a business perspective that’s not acceptable. | Structuring | Because permission is involved when different roles act on different components, it is categorized as structuring. |
| That’s a normal way I’ve seen it done so that’s how I would think that should be done. | Problem comprehension | This is a problem comprehension because the representing system is created based on other experience. |

The coding process was done by a single rater who collected all 26 protocols, read the transcriptions, examined the design documents and watched the recorded videos. Because the rater could recognize who the person was by their voice and handwriting, the coding process was not a blind review. The rater was also aware of the skill levels (novice or expert) of the participants, by whom the protocols were generated, during the scoring process.

When Chi (1997) discussed interrater reliability issues with regard to the verbal analysis methodology, she mentioned three problems that needed to be addressed:

The first concerns how carefully one has to define the categories a priori, before each rater codes the data. The second has to do with whether discrepancies between two coders should always be resolved. If so, the third issue has to do with when discrepancies should be resolved. (p. 306)
Because there was only one rater, the second and third problem was not relevant to this study. Instead, the main concern was how consistent the rater was in coding the episodes.

After the coding process was done, one protocol was randomly selected from the 26 collected protocols a month later for a repeat of the coding process by the same rater. This protocol had 58 episodes, and 46 episodes were coded the same categories as previous. The reliability was 0.793 in this case. Therefore, the rater was pretty consistent when coding the episodes.

With regard to the first problem about the definition of the categories, three protocols, randomly selected from 26 protocols, were rated by two other independent raters. The reliability scores for the first rater were: 0.571, 0.384, and 0.483; the reliability scores for the second rater were: 0.60, 0.411, and 0.283. Many of the discrepancies came from ambiguous statements such as “I think I am done with the design” that can be considered non-design (the rater of this study) or problem comprehension (other raters). Although the reliability scores were not impressive and some were low, which indicated that the definition of the categories might need work in order for other researchers to duplicate this study, the results of this study should not be discounted because they were based on the single rater who was consistent about eighty percent of the time.

**Summary**

In this chapter, the research methodology employed in this study to uncover the insights of the proposed research questions, is described and discussed. Because understanding cognitive processes is a task involving the estimation of human cognitive activities, this study used an exploratory approach to examine novice programmers’ verbal protocol during a task of software design. Eight novice programmers, who are undergraduate students at The Pennsylvania State University, participated in this study. Their cognitive processes of design were collected by
following think-aloud protocol (Ericsson & Simon, 1984) and analyzed through verbal analysis (Chi, 1997).
Chapter 4

RESULTS OF THE RESEARCH

“The research embodying the verbal analysis method focuses on learning. More specifically, its goal (perhaps not yet achieved) is to capture the representation of knowledge that a learner has and how that representation changes with acquisition.” (Chi, 1997, p. 274)

In this chapter, verbal protocols from both experts and novices are presented and analyzed quantitatively and qualitatively from the standpoint of their behaviors and cognitive processes. Results from experts’ data are presented first, followed by those from the novices.

At the end of each novice case, a summary is presented to describe the progress made by the individual over the course of a semester. This summary compares and contrasts cognitive processes, and posits the relationships between mental models and those cognitive processes. This chapter ends with a short summary of results that answers the research questions posed in chapter one.

Experts’ Data

Demographic Data

Expert E1 was one of the instructors for IST240 and a Ph.D. candidate in IST at The Pennsylvania State University when the data was collected. He received his master’s degree in software engineering and had spent eight years in industry as a software engineer, designing and developing software for Fortune 500 companies, startups, and small organizations.
Expert E2 was another IST240 instructor and a Ph.D. candidate in IST at The Pennsylvania State University when the data was collected. He was a software engineer before attending IST for his doctoral degree, and he has been writing computer programs for more than 10 years.

Design Behavior

It seemed, from the time spent on the task, that E1 and E2 read the design scenario and the instructions thoroughly and carefully. Their first design activity after reading the instructions, was for each to represent his understanding of the system by creating either a context diagram or a task list. These artifacts acted as a mapping mechanism from the represented world (the design scenario) to the representing world (the mental model), and became references and checklists for the experts during their design process. These artifacts also replaced the role of the instructional text for these participants because it was easier for experts to reference their own representations or memory than the original instruction.

In the task list created by E2, entities (e.g., patron, librarian) and their respective tasks were presented in a hierarchical structure. It was, in essence, a checklist or progress benchmark to tell what had been done and what still needed to be done. E1 created a context diagram that had both entities in blocks and tasks in lines to help him see the system graphically. With the help of those artifacts, they could quickly map their mental model to the requirements and also monitor their design progress.

After the new problem representation was created, the experts started to examine the details and functions for each entity and the relationships between entities. Even though implementation (coding) was not required, many programming specific properties (such as Java classes, concepts of methods and attributes, object-oriented concepts) were utilized in the design
process, especially when the experts had defined the most elementary components of the tasks. For example, when an entity was identified as being necessary to create, experts wrote down both the attributes of the entity and the methods needed to manipulate the entity as if designing Java classes. In addition, experts wrote function calls in a style that was similar to the Java methods being used. The language syntax of Java programming was heavily used in the experts’ design documents.

When entities with attributes and methods were created and ready to be used, E1 created a state diagram for a book owned by the library. States of a book included: shelved, checked out, reserved, overdue, discarded, lost, returned, and so on. He commented on creating the state diagram later in the design process:

This [the state diagram] is going to be incredibly useful, as [the] system evolves. State diagrams [are] very, very important, especially if you are tracking [a] physical object. State diagrams are essential…state diagrams have a very long life in terms of their usefulness. (From expert E1)

By tracing a book’s use, each possible state during the book’s useful life was identified and written down on a paper. The actions that would bring a book from one state to another were also sequenced to connect the related status. This state diagram was useful to E1 because several states and functions which could not be directly inferred from the instruction were discovered during the process of generating this diagram.

Another representation that helped tremendously was the transition diagram created by E1 and E2, with E2 using it intensively. A transition diagram could be described as two or more entities with an arrow pointing from one to another. Each arrow denoted an action. The starting of the arrow represented the initiator of the action and the ending of the arrow represented the receiver. For example, to represent a librarian editing an item, the first arrow, representing an editing item request, would go from librarian to system. The second arrow, representing asking an item ID, would go from system to librarian. Editing an item was accomplished with a series of
information exchanges in such a fashion. Both experts created several transition diagrams when they were in the process of functional decomposition.

Cognitive Processes

Table 4.1 and 4.2, are quantitative representations of the experts’ cognitive activity episode frequency. The type of cognitive processes, number of episodes attended, and the percentage of each category during the entire design process are listed in columns one, two, and three.

Decomposition was the most frequently attended cognitive activity for both experts, who employed it for both problem decomposition and procedural decomposition. Because of their experience, experts were capable of detailed procedural decomposition for each task.

Different design strategies led to a difference in the cognitive processes being attended to by each expert. E1 used multiple design tools, such as context diagramming, state diagramming and transition diagramming in the design process. On the other hand, E2 used a task list and transition diagrams in his design process. Because creating both task lists and transition diagrams require decomposition activity, Figure 4.1 and 4.2 depict the number of episodes and percentage of episodes by category for both E1 and E2.

Table 4.1

Cognitive Process Frequency by Expert E1

<table>
<thead>
<tr>
<th>Category</th>
<th>Frequency</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Problem Comprehension</td>
<td>14</td>
<td>9.21%</td>
</tr>
<tr>
<td>Decomposition</td>
<td>45</td>
<td>29.61%</td>
</tr>
<tr>
<td>Structuring</td>
<td>19</td>
<td>12.50%</td>
</tr>
<tr>
<td>Mental Simulation</td>
<td>26</td>
<td>17.11%</td>
</tr>
<tr>
<td>Evaluation</td>
<td>15</td>
<td>9.87%</td>
</tr>
</tbody>
</table>
Table 4.2

Cognitive Process Frequency by Expert E2

<table>
<thead>
<tr>
<th>Category</th>
<th>Frequency</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Problem Comprehension</td>
<td>13</td>
<td>15.66%</td>
</tr>
<tr>
<td>Decomposition</td>
<td>39</td>
<td>46.99%</td>
</tr>
<tr>
<td>Structuring</td>
<td>5</td>
<td>6.02%</td>
</tr>
<tr>
<td>Mental Simulation</td>
<td>9</td>
<td>10.84%</td>
</tr>
<tr>
<td>Evaluation</td>
<td>4</td>
<td>4.82%</td>
</tr>
</tbody>
</table>

Figure 4.1 Comparison of experts’ design cognitive processes by the number of episodes.
Scatter plots provide another perspective in analyzing cognitive processes. With the scatter plots, one can examine the chronological occurrence of an expert’s cognitive processes. Figure 4.3 shows that problem comprehension is condensed toward both ends of the whole design activity. The data indicates that as soon as E1 generated a preliminary mental mode, he then turned to decomposition, structuring, and mental simulation until later in the process when he did a recheck of the system. Furthermore, because mental simulation did not occur until halfway through the design task, that coincided with my initial intuitive interpretation that designers usually need to generate a draft of the design before mental simulation can take place.
Figure 4.4 A scatter plot representing E2’s chronological design processes.

According to Figure 4.4, expert E2 had a similar behavioral pattern to expert E1 in the problem comprehension activity. When E2 started using transition diagrams to decompose tasks, most episodes were coded as decomposition.

The major difference between E1 and E2 is in the category of evaluation. E1 did many evaluations throughout the design process and they were evenly distributed, whereas E2 did not do as many. In fact, evaluation was the least frequent of E2’s cognitive activities. This did not necessarily indicate that E2 had limited cognitive capability in evaluating his design, however. The definition of evaluation is to resolve difficulties by comparing alternatives and making decisions. E2, in fact, did not seem to encounter difficulties in his design process, and that might have reduced the need for evaluation.

Qualitative Analysis

It is obvious that during the problem comprehension phase, E1 and E2 not only read the description, but also they had their own problem interpretations in their working memories and then proceeded by creating the interpretation on the paper. Most importantly the experts were able to generate a robust structure early on in the problem comprehension phase, even though the
structure may not be explicit in the instructional text. After comprehending the design scenario, both experts were capable of creating a new representation without much modification throughout the design, which suggests that their initial mental representations of the system were close to the finished design. In short, the experts had no problems at all in generating a complete mental representation in a short period of time.

In the meantime, part of the decomposition was also being done during the problem comprehension phase, while the initial mental representation was being created. That is, sub-components were parsed to make a context diagram or a task list, but the implementation of detailed function was missing. Many of the relationships and interactions among the subcomponents, however, were unknown or incomplete at this point and, their operation could not be simulated mentally. As a result, it was unclear whether the designed system was workable or whether there was room for improvement. At this moment, the design scenario was well understood and decomposed into at least one level down from the top of the initial state of the problem space.

When all sub-components were in place, the experts started to connect them by examining relationships, interactions, and functions to accomplish the overall goal. Functional design relied primarily on decomposition and relationships and interactions being attended to by cognitive processes of structuring and mental simulation. Experts, with their programming experience, were very good at functional design. They seemed to be able to attend to all of the required information from their LTM. There were only a few times that the experts had to perform a restructuring, and only then because they recognized a similarity and wanted to reuse it.
Novices’ Data

In this section, each participant’s three design sessions will be presented, compared, and contrasted. Novice programmers demonstrated a variety of design strategies and approached design from various aspects. Therefore, the total number of episodes varies from person to person, making it less meaningful to discuss cognitive activity based on tally alone. That is, the inferences from comparing episodes among novices would not be as informative as it was for the experts.

Instead, what is meaningful is the development across novices’ episodes, because the analysis of verbal data will provide insights about a novice’s cognitive progress. The discussion in this section, hence, will focus on the changes among sessions. In the case of a dramatic difference in design behavior and cognitive process, verbal analysis from the perspective of cognitive development and learning helps to locate the source and reason for those changes as they occurred during the period of data collection.

In the following sections (categorized by the participants’ pseudonyms), each participant’s demographic data will be presented briefly first, followed by analysis of the three design sessions, then concluded by an analysis made from all three design sessions.

Benjamin

Demographic Data:

Benjamin, who was 20 years old when participating in this study, was an Information Sciences and Technology (IST) major, third-year undergraduate student. He had 15 years of experience using computers for checking emails, typing school work, playing video games, and
designing web sites. He had experience using C++ and PHP, gained as part of the IST curriculum and had done a group project in PHP (a scripting language) for an IST 210 course.

**Design Session I:**

When asked to create a system responding to the software design scenario, Benjamin started by designing a user interface (UI) and divided the users to two categories, the librarian and the patron, as the instruction sheet implied. He then started to create web pages by going through each of the tasks that were listed on the instructional brief with the scenario (Appendix B). During the design process, he mentioned general UI design concepts and also tools and environments that could be used in designing this type of application.

You can do that in visual basic on a, on a computer. VB, with a VB … ASP.net I think is the, is the web application Microsoft uses. You can do a lot of open source things too like AJAX or Python, Java Script.

The most frequent attended cognitive activity was *non-design* in Benjamin’s first design session (Table 4.3). He would wonder aloud about general design issues, and those activities were coded as non-design, hence the high percentage of non-design episodes.

Table 4.3

Cognitive Process Frequency by Benjamin in Design Session I

<table>
<thead>
<tr>
<th>Category</th>
<th>Frequency</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Problem Comprehension</td>
<td>9</td>
<td>16.98%</td>
</tr>
<tr>
<td>Decomposition</td>
<td>16</td>
<td>30.19%</td>
</tr>
<tr>
<td>Structuring</td>
<td>16</td>
<td>30.19%</td>
</tr>
<tr>
<td>Mental Simulation</td>
<td>10</td>
<td>18.87%</td>
</tr>
<tr>
<td>Evaluation</td>
<td>2</td>
<td>3.77%</td>
</tr>
</tbody>
</table>
In terms of cognitive activity related to software design, first of all, there was no systematic procedure in the design process based on the analysis of his verbal protocol. Because he would often deviate from one task to another, his initial mental model was brittle, requiring new information to be added throughout the design process. Therefore, he restructured and expanded his mental model through the design process. In the second half of Figure 4.5 we can see that mental model episodes occurred more frequently than structuring and decomposition. That these scenario simulations occurred ahead of decomposition and structuring also indicates his mental simulation was expanding while he designed the system.

Although there were decomposition episodes, none of them was a procedural decomposition. As a result, all mental simulations were scenario simulations rather than functional simulations. So Benjamin had a declarative knowledge of the system, but not a practical knowledge, which prevented him from performing procedural decomposition and functional simulation (Ohlsson, 2008).

Figure 4.5 A scatter plot representing Benjamin’s chronological design processes in design session I.
In design session two, Benjamin stated many ideas of what could be done, rather than what should be done for the system. For example, he mentioned two possible applications of UI for patrons:

First one is a kiosk system where a kiosk, you know a physical kiosk, where the patron would just go up and there will be touch screen or a keyboard and a mouse. … and on this, it would basically be a big bubble about that size that the patron can simply touch…You take your library card, swipe in…it’s like an ATM machine, you have a series of buttons through that you can do…Now something else you could try to do is, have you seen Microsoft surface? You could do something along those lines. So have a table, like a coffee table where the patrons can sit down and there can be reading materials actually on the table. This is what you can go a number of ways. You can swipe in your card and, you know, what you’ve checked out lately actually comes up…can click one and move them around, or open them up or someway and, and rotate them. And then read through them like that.

Although those were legitimate design principles, they were not directly related to the design task. Again, non-design episodes were the most frequent for him in design session two (Table 4.4). Among other design cognitive activities, episodes in decomposition, structuring, and mental simulation were balanced. However, the mental simulations were scenario simulations which, again, indicated his mental model was declarative rather than practical. As a result, his design was very user-centered as he concentrated heavily on UI design and usability issues. A lot of his design concepts were drawn from his personal experience of using technologies.

…like in a cell phone, you could just have a list of numbers as the phonebook. Or you can have a series of pictures of people and you click on the pictures and the pictures bring you to the person’s name, their information, their address, their phone number, their email addresses, um, maybe some notes where they work. That’s a lot more useful than just a phone number.

<table>
<thead>
<tr>
<th>Category</th>
<th>Frequency</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Problem Comprehension</td>
<td>2</td>
<td>6.25%</td>
</tr>
</tbody>
</table>
According to Figure 4.6, there was a hierarchy formed from problem comprehension, decomposition, structuring, and mental decomposition, in which the first occurrence of lower cognitive activities (problem comprehension) appeared earlier than the higher ones (evaluation). Intuitively, this is a logical sequence in a design task. Then again, they did not produce a pattern, which attests to Benjamin’s unsystematic design behavior. Similar to his first design session, Benjamin’s design behavior was opportunistic and his mental model was a UI-based one. In addition, his decomposition activity was procedural decomposition. The sub-components were organized based on anticipated user behavior and system response.

To summarize this section, the verbal analysis indicated that Benjamin’s mental model was formed from his personal experience and observation. This likely was due to his lack of detailed design in that experience.
Design Session III:

Table 4.5
Cognitive Process Frequency by Benjamin in Design Session III

<table>
<thead>
<tr>
<th>Category</th>
<th>Frequency</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Problem Comprehension</td>
<td>8</td>
<td>14.04%</td>
</tr>
<tr>
<td>Decomposition</td>
<td>17</td>
<td>29.82%</td>
</tr>
<tr>
<td>Structuring</td>
<td>18</td>
<td>31.58%</td>
</tr>
<tr>
<td>Mental Simulation</td>
<td>10</td>
<td>17.54%</td>
</tr>
<tr>
<td>Evaluation</td>
<td>4</td>
<td>7.02%</td>
</tr>
</tbody>
</table>

Figure 4.7 A scatter plot representing Benjamin’s chronological design processes in design session III.

In the third design session, Benjamin was able to concentrate on the design task more than in the previous two, as one might infer from the shrinking number of non-design episodes. Decomposition and structuring episodes accounted for more than sixty percent (61.4%) of total design-related episodes, according to Table 4.5. Among all categories, structuring was the most frequently attended cognitive process, and evaluation was the least frequent. Except for one mental simulation episode that occurred in episode six, the sequence of first occurrence of design cognitive process was problem comprehension, decomposition, structuring, and mental simulation. Excluding episode number six where Benjamin was responding to the experimenter’s
inquiry, the pattern of occurrence of cognitive process seemed to match with intuition and also the experts’ behavior.

With regard to a mental model, Benjamin again incorporated many of his experiences in forming a mental model of the system, but he also included some Java programming knowledge in his design.

And, and that would be the primary key. And on top of that, we’ll have those software methods going on top of that. You know, in my Java class we talked about that and, you know, you have object, you have methods that act on those objects. And this would be something similar to that.

There was a unique perspective that separated his design from the instruction: he did not include the librarian in any part of the system, all transactions in Benjamin’s system were between books and patrons.

“When I think of that stuff, I just think of the patron and the books, I don’t really think of the librarians going into to that because we are really just dealing here with the patron themselves and then the books themselves. The person who is actually doing these tasks, it doesn’t really matter.”

This was a huge difference considering that the instruction implied (without any specific intent from me) there were two sets of tasks: tasks for the patrons and tasks for the librarians. Benjamin’s mental model only focused on the patrons, the books, and the interactions between them. The role of librarian was either ignored or merged into the transactions. Overall, his mental model was still very user-centered and UI-based while slowly improving (considering that his recently transferred knowledge was transferred only from a course in computer programming).
Summary

Figure 4.8 Comparing Benjamin’s three design sessions by the number of design-related episodes.

Figure 4.9 Comparing Benjamin’s three design sessions by the percentage of design-related episodes.

From Figure 4.8 and 4.9, one of the changes in cognitive process that Benjamin demonstrated was the shrinking percentage of non-design activity. During this semester, his mental model apparently evolved into a more complex and concrete one. He was able not only to evaluate different options and give rationales, but also to execute sophisticated mental simulation. Learning Java programming might have helped him see the underlying functionalities of each task, improving his procedural decomposition, and enabling his ability to organize sub-
components. Although it seems there is not much difference in decomposition and structuring in Figure 4.8 and 4.9, the quality of those cognitive processes did improve. In terms of decomposition, he had improved from centering on UI to adding some of his own understanding of the design scenario. In structuring cognitive process, Benjamin also improved from making connections based on prospective users’ behavior to linking different components to accomplish a task. In order to make both of these improvements possible, he had to be able to think of the system in terms of the components that support the UI features, rather than just the UI features themselves.

Benjamin’s mental model changed from very self-centered and user-interface-oriented to more function-centered after learning computer programming. It is likely that programming knowledge enables those changes. What did not change was applying knowledge from other domains to his design.

Caleb

Demographic Data:

Caleb, who was 20 years old when he participated in this study, was an IST major, third-year undergraduate student. He had eight years of experience using computers for playing video games, browsing the Internet, reading news, connecting with friends, checking emails, and doing school work. In terms of computer programming, he had experience using Visual Basic as part of the IST curriculum and had done an individual project (a dice rolling game) in Visual Basic as a course requirement. Caleb had also completed the IST210, a course on database design.
After reading the instructions, Caleb started off by designing components without creating a visible artifact to represent his mental model. Tasks were then divided into two groups, one for the book and the other for the patron, and each part was looked at independently. Because he did not generate an external model on paper, his mental model was formed and stored in his memory. Tasks were examined one by one and assigned to either the patron or the librarian. According to table 4.6, decomposition and structuring accounted for 90.91% of the total episodes, in which decomposition alone took up 57.58% of the total episodes. This suggests that Caleb was simply listing and organizing tasks that could be found in the instructions.

The lack of a problem comprehension episode might indicate that Caleb used his prior experience to form his mental model so that he did not need to rely on the instructions. Caleb’s mental model must have been very simple because there was only one mental simulation episode, no evaluation episode, and a relatively short design session.

<table>
<thead>
<tr>
<th>Category</th>
<th>Frequency</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Problem Comprehension</td>
<td>2</td>
<td>6.06%</td>
</tr>
<tr>
<td>Decomposition</td>
<td>19</td>
<td>57.58%</td>
</tr>
<tr>
<td>Structuring</td>
<td>11</td>
<td>33.33%</td>
</tr>
<tr>
<td>Mental Simulation</td>
<td>1</td>
<td>3.03%</td>
</tr>
<tr>
<td>Evaluation</td>
<td>0</td>
<td>0.00%</td>
</tr>
</tbody>
</table>

Caleb’s design strategy was focusing on one part of the system, the book flow, and creating necessary functions for it. He then switched to another part of the system, the patrons, and created functions to support patrons’ tasks. After both were created, additional functions were added and connected based on these two separate parts. Additional functions were those which
could be shared by or needed to reference data from both parts. Caleb stopped his design when he felt he had accomplished all functionalities. The extremely low number of mental simulation and evaluation episodes reflected his design behavior as he did not spend effort to ensure the applicability of his design to the scenario. One of his verbal protocols did suggest that he embraced a divide-and-conquer strategy, when he said “I’ll do the patron for now and tie them both together.”

Because a lot of Caleb’s cognitive activities revolved around decomposition and structuring, Figure 4.10 shows that he decomposed the problem into several sub-components and then connected them together; then he repeated the decomposition and structuring processes several times, until the design was finished. Two episodes of problem comprehension occurred early in the design process, suggesting that he used either his mental model or his experience in the process of comprehending the system.

<table>
<thead>
<tr>
<th>0</th>
<th>5</th>
<th>10</th>
<th>15</th>
<th>20</th>
<th>25</th>
<th>30</th>
<th>35</th>
<th>Episode</th>
</tr>
</thead>
<tbody>
<tr>
<td>Problem Comprehension</td>
<td>❧</td>
<td>❧</td>
<td>❧</td>
<td>❧</td>
<td>❧</td>
<td>❧</td>
<td>❧</td>
<td>❧</td>
</tr>
<tr>
<td>Decomposition</td>
<td>❧</td>
<td>❧</td>
<td>❧</td>
<td>❧</td>
<td>❧</td>
<td>❧</td>
<td>❧</td>
<td>❧</td>
</tr>
<tr>
<td>Structuring</td>
<td>❧</td>
<td>❧</td>
<td>❧</td>
<td>❧</td>
<td>❧</td>
<td>❧</td>
<td>❧</td>
<td>❧</td>
</tr>
<tr>
<td>Mental Simulation</td>
<td>❧</td>
<td>❧</td>
<td>❧</td>
<td>❧</td>
<td>❧</td>
<td>❧</td>
<td>❧</td>
<td>❧</td>
</tr>
<tr>
<td>Evaluation</td>
<td>❧</td>
<td>❧</td>
<td>❧</td>
<td>❧</td>
<td>❧</td>
<td>❧</td>
<td>❧</td>
<td>❧</td>
</tr>
<tr>
<td>Non-design</td>
<td>❧</td>
<td>❧</td>
<td>❧</td>
<td>❧</td>
<td>❧</td>
<td>❧</td>
<td>❧</td>
<td>❧</td>
</tr>
</tbody>
</table>

Figure 4.10 A scatter plot representing Caleb’s chronological design processes in design session I.

**Design Session II**

In this design session, Caleb, again, started from creating a diagram for a librarian and required tasks for the librarian. He then created a diagram for a patron and all the required tasks for the patron. After both diagrams were done, he started to look for tasks that interacted between
the patron and the librarian. A self-reviewing process was also conducted to make sure nothing was missing from the requirements of the instructional brief, which did not happen in design session one. Decomposition and structuring were the major processes in the first half of the design. When he moved to the self-review process, mental simulation started to occur intensively. There were a few exceptions when he found problems and had to fix them. Table 4.7 shows that Caleb’s cognitive processes were centered on decomposition, structuring, and mental simulation.

One interesting observation is that, compared to design session one, he spent about the same amount of episodes in decomposition and structuring (30 episodes for the first session and 33 episodes for the second session). So the difference between the first and second design session was that Caleb added a self-review process that, in turn, increased mental simulation episodes. Figure 4.11 shows that most mental simulation episodes are in the second half of the design.

Table 4.7
Cognitive Process Frequency by Caleb in Design Session II

<table>
<thead>
<tr>
<th>Category</th>
<th>Frequency</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Problem Comprehension</td>
<td>4</td>
<td>6.90%</td>
</tr>
<tr>
<td>Decomposition</td>
<td>18</td>
<td>31.03%</td>
</tr>
<tr>
<td>Structuring</td>
<td>15</td>
<td>25.86%</td>
</tr>
<tr>
<td>Mental Simulation</td>
<td>21</td>
<td>36.84%</td>
</tr>
<tr>
<td>Evaluation</td>
<td>0</td>
<td>0.00%</td>
</tr>
</tbody>
</table>
Because there was no external model created, it might reasonably be assumed that Caleb implemented his mental model directly into the design. In addition because he modified his design often during the self-reviewing process, his mental model was incomplete or had flaws.

During the self-review process, he said:

...because when you are trying to check something in or out, you need the same information whether it's who it is or what it is so there is no point in changing, or having two separate functions for that.

If he had had a complete working mental model, he might have realized this potential when he first wrote his design on paper. As a result, Caleb’s mental model initially was a replica of the instructions, and he restructured his mental model during the design process. Again, there was no evaluation activity in second design session.

**Design Session III**

Table 4.8

Cognitive Process Frequency by Caleb in Design Session III

<table>
<thead>
<tr>
<th>Category</th>
<th>Frequency</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Problem Comprehension</td>
<td>7</td>
<td>12.07%</td>
</tr>
<tr>
<td>Decomposition</td>
<td>18</td>
<td>31.03%</td>
</tr>
</tbody>
</table>
In this design session, Caleb started with the librarian and all tasks to be accomplished by the librarian. He used decomposition processes, then structuring processes, followed by mental simulation. Then he moved to the patron, following the same model. He connected these seemingly independent portions of the design together using the tasks that required information from both the patron and the librarian. After the design was done, he used a diagram to represent his design for clarity by saying, “I am just going to try to draw out the basic layout of the title head so I get a better mental image of it all.” The scatter plot, Figure 4.12 captured some of his design behavior: decomposition, structuring, and mental simulation for the librarian, then decomposition, structuring, and mental simulation for the patron, and then mental simulation for the diagram during a self-reviewing process.

As in previous design sessions, Caleb did not create any external model in the beginning. He recalled the information from his memory instead. In one instance, he was able to capture the similarity of two tasks when he first wrote them down and merged them together.

Um, well, actually the renew can be added in there as well because it will be just changing the availability for a longer due date or make certain period of time. So it doesn’t need something separate than in and out.
However, later in the self-reviewing process, he did find other pairs of tasks that he could potentially merge, further reducing redundancy. Caleb’s mental model was closer to what his final design became, and he showed fewer modifications in this session compared to the previous two designs, although there still were some. Also his attempt to use the diagram later in the design process did not seem to be effective because he kept changing the arrows and it started to become complicated and he seemed to be stuck at times. Nevertheless, he managed to use the diagram to assist his mental simulation, and modified areas of the design he thought to be confusing.

Summary

![Bar chart comparing Caleb's three design sessions by the number of design-related episodes.](image)

Figure 4.13 Comparing Caleb’s three design sessions by the number of design-related episodes.
Figure 4.13 and 4.14 show that the biggest difference for Caleb in a semester of learning programming was his ability to use mental simulation to improve his design. Compared to design session one, mental simulation episodes increased dramatically in design session two and three. There is, however, no evidence that Caleb lacked mental simulation ability in the beginning of the study, just that mental simulation processes were often used to ensure the correctness of a design and, consequently, to improve the design.

In the first design session, Caleb stopped his design without a self-reviewing process. There were 34 episodes in the first design session, 58 and 60 for the second and third design sessions. The analysis, both quantitatively and qualitatively, of his verbal data suggests that he did not engage in a mental simulation process in the first design session because he did not attempt to self-review. The mental simulation in his self-reviewing process did change his design and his mental model.

Caleb also followed a strategy that was to decompose the system into two sub-components, and then design for each one of them. Any tasks that required both sub-components to interact were designed after both sub-components were done in isolation. He used this strategy in every design session with only a slight variation: no mental simulation in design process one,
mental simulation after both sub-components were finished in design session two, and mental simulation after each sub-component was finished and also when both subcomponents were finished in design session three. More self-reviewing process, however, does not increase the number of episodes of mental simulation accordingly. This could mean Caleb felt that he did not need to mentally simulate certain areas of the design once he had confidence that those parts would function properly.

Furthermore, his mental models were becoming more robust and less redundant during the semester. In design session two, he recognized two functions could be combined after his review of the structure of his design. In design session three, he was able to recognize their similarity right away. Manipulating these abstract objects in memory and structuring the objects and their relationships without the assistance of a visual representation or external model is a notable improvement.

**Jedediah**

**Demographic Data:**

Jedediah, who was 19 years old when participating in this study, was an IST major, second-year undergraduate student. He had more than 10 years of experience using computers for checking emails, writing papers, researching for papers, and playing video games. He had experience using C++ as part of the IST curriculum. In addition, he also used HTML and JavaScript, which he taught himself. Other than writing small Java Applet applications, Jedediah had no prior experience in any software design project. When asked about how he would approach a design project, he specified that he would diagram first on a high level, and then get
into details, starting from the top down. He also noted that he was not taught by anyone about the
top-down strategy; it just seemed to be the most logical way to approach a design project.

**Design Session I:**

In this design session, Jedediah started by creating a task list from reading the
instructional brief so that the list could be used to check against his design later. This task list
represented his first mental model of the system, and it had two categories with their respective
tasks listed under each category. It appears that, in the mental model, each task was seen as an
independent sub-component at this point, because Jedediah not only did not draw any symbols to
indicate the connections between these tasks, but also checked them off one by one. After the
task list was manufactured, he created a diagram that contained three database tables for storing
data, and connected the tables by how the data flowed from one table to another. With this
diagram in hand, he was able to mentally simulate the tasks that he listed previously, and used
this as a way of checking whether the database tables could facilitate those tasks. The last thing
he did for the design was to generate user interfaces and to describe how items on the user
interface would use both the tasks on his list and the database diagram to accomplish required
tasks.

Table 4.9

**Cognitive Process Frequency by Jedediah in Design Session I**

<table>
<thead>
<tr>
<th>Category</th>
<th>Frequency</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Problem Comprehension</td>
<td>3</td>
<td>6.38%</td>
</tr>
<tr>
<td>Decomposition</td>
<td>18</td>
<td>38.30%</td>
</tr>
<tr>
<td>Structuring</td>
<td>7</td>
<td>14.89%</td>
</tr>
<tr>
<td>Mental Simulation</td>
<td>14</td>
<td>29.79%</td>
</tr>
<tr>
<td>Evaluation</td>
<td>5</td>
<td>10.64%</td>
</tr>
</tbody>
</table>
Table 4.9 shows that decomposition and mental simulation were the two most often attended cognitive processes. The reason for more mental simulation than structuring episodes was that some of the mental simulations were simulating the functions on the task list. In the meantime, on the scatter plot (Figure 4.15), one can see that there were decompositions followed by mental simulation before structuring occurred. There were also “decomposition, structuring, mental simulation” triplets when he designed tasks that crossed over to connect two entities, such as a patron and librarian or patron and book.

When Jedediah designed the system, he mentioned twice that some areas were beyond his ability. The first occasion was his comment about programming, and the second one was about accessing a database through scripting language.

But as far as doing it remotely I’m not really quite sure how that could be done. I guess you put all the databases on line, then you ran a whole bunch of scripts, you could probably access data from those and change records and what not. But that’s way beyond my abilities for sure.

His recognition of inability did not affect his ability to advance his design activity, however. He knew what needed to be done without knowing how to actually do it. The merit of this observation is the realization of the possibility for someone who does not have deep programming skills to acquire design knowledge.
**Design Session II:**

In this design session, Jedediah created a user interface and listed tasks for a librarian. After the first “welcome splash screen” was created, he then focused on each individual task in turn and created a UI for each task if he found that necessary. Patron tasks were designed following the same strategy. The UIs became transition diagrams in Jedediah’s design process. Through those user interfaces, the data flow for each task became clear for him, and he was able to envision the connections of the system through users’ perspectives. The disadvantage of using this method, from the standpoint of design, is redundancy. Tasks for the user can look different but, in fact, they share the same concept or manipulate the same set of data. For example, he said:

> So the first page would be check in, check out…well, maybe it would just be better, rather than combining those two, have like separate them…that way, depending on what kind of permissions, maybe only certain librarians are allowed to check in books, only certain librarians are allowed to check out books.

In terms of design, “check in” and “check out” potentially can be combined. His approach led him to separate them because the design isolated each user’s perspective from that of another.

<table>
<thead>
<tr>
<th>Category</th>
<th>Frequency</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Problem Comprehension</td>
<td>5</td>
<td>6.25%</td>
</tr>
<tr>
<td>Decomposition</td>
<td>29</td>
<td>36.25%</td>
</tr>
<tr>
<td>Structuring</td>
<td>16</td>
<td>20.00%</td>
</tr>
<tr>
<td>Mental Simulation</td>
<td>25</td>
<td>31.25%</td>
</tr>
<tr>
<td>Evaluation</td>
<td>5</td>
<td>6.25%</td>
</tr>
</tbody>
</table>
Table 4.10 shows that in this design session, Jedediah’s most frequently attended cognitive activities were decomposition and mental simulation, similar to the distribution of activities in design session one. From verbal analysis, the scenario of using the library took control of the construction of his mental model which, consequently, had more of the structure of a tree with two major trunks rather than a network. Each task branched out from the center of the leaf. The need for connecting to other tasks was, therefore, less significant and less important. Jedediah’s design pattern, as shown in Figure 4.16, is opportunistic because cognitive activities disperse without a logical pattern.

**Design Session III:**

Jedediah used the same strategy as he did in design session two where he started with designing UIs for both patrons and librarians. Tasks for both types of users were added by simulating what needed to be done later. Each branching out was rather linear. Because this strategy built his mental model from what each type of user would see and do, his attempts to group similar functions were unsuccessful.

The next thing I will do would be to group the functions together that run off similar algorithms... So things that will go together would be, um, for patrons would be look up personal information would pair up with, um, ha, ha, that one wouldn’t pair up with anything. Um, look up library books, um, I don’t know...
He tried to find tasks that were similar to looking up personal information and failed. He then tried to group “look up library books” with other tasks and was unsuccessful again. With these two unsuccessful attempts, he gave up the grouping idea and proceeded with what he did in design session two. The redundant and linear results he achieved are to be expected when mental models are constructed by tracing scenarios. Again, Jedediah spent at least 28.24% of episodes on thinking and talking about UIs, which was coded as non-design activity as showed in Table 4.11. He also mentioned several general principles such as usability and consistency which indicated his mental model was very user-oriented.

I would go with the graphic user interface again. Um, because library patrons aren’t exactly the most technological bunch…even call it something simple like ‘find the book’, so that people immediately know what it is…it will keep it consistent across different functions. So once again, since the patrons are probably not the most technology savvy, most of them, um, inconsistency will probably be very frustrating. So keeping it consistent, keep everyone happy.

Among cognitive processes that were related to design, problem comprehension and mental simulation were again the two most frequently attended categories. It indicates his mental process has not changed much across design sessions. Figure 4.17 shows that Jedediah used decomposition, structuring, and mental simulation, but without a pattern. He would often jump from one activity to another. In this type of design behavior, a visible design aid such as a task list or transition diagram might have helped. It can reduce the designer’s cognitive load by offloading the mental model to an external model and keeping track of current design progress.

Table 4.11
Cognitive Process Frequency by Jedediah in Design Session III

<table>
<thead>
<tr>
<th>Category</th>
<th>Frequency</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Problem Comprehension</td>
<td>8</td>
<td>13.11%</td>
</tr>
<tr>
<td>Decomposition</td>
<td>23</td>
<td>37.70%</td>
</tr>
<tr>
<td>Structuring</td>
<td>13</td>
<td>21.31%</td>
</tr>
<tr>
<td>Mental Simulation</td>
<td>15</td>
<td>24.59%</td>
</tr>
</tbody>
</table>
Summary

Figure 4.18 shows that there is an increase in problem comprehension in terms of the number of episodes. Analysis of Jedediah’s verbal protocols, mental model, and design strategies, indicates that the discrepancy may come from the use of a task list. In the first design session, he used a task list before he started other cognitive design activities. This behavior can also be seen on the scatter plot chart (Figure 4.15, 4.16, and 4.17). When he used a task list, he mitigated the cognitive load, which allowed him to concentrate on other design tasks. On the other hand, in design sessions two and three, his strategy of walking through UIs hindered his ability to offload problem comprehension to an external model. For both design sessions two and three, problem comprehension episodes appeared later in the design process. Nevertheless, the percentages of problem comprehension are not very different (Figure 4.19). Using a task list affected only when problem comprehension occurred, but not the frequency of occurrence.
Figure 4.18 Comparing Jedediah’s three design sessions by the number of design-related episodes.

Figure 4.19 Comparing Jedediah’s three design sessions by the percentage of design-related episodes.

Figure 4.19 also shows that when the design method changed, the percentages of decomposition, mental simulation, evaluation, and non-design activity changed as well. Figure 4.19 suggests that some portion of decomposition, mental simulation, and evaluation were switched to non-design. When Jedediah was using a UI-based mental model, his attention was drawn to usability and consistency. UI appearance became important and preoccupied his cognition. As a result, other types of cognitive activity were reduced.
David

**Demographic Data:**

David, who was 21 years old when participating in this study, was an IST major, third-year undergraduate student. He had seven years of experience using computers for reading emails, receiving information about school and his community, and as a tool for learning more about computing in general. He also had experience using C++ as part of the IST curriculum and created a black jack game using C++ for the final project in the CMPSC101 class. In addition, he was involved in a group project to design a database supporting an online book store for the IST210 class.

**Design Session I:**

Table 4.12

Cognitive Process Frequency by David in Design Session I

<table>
<thead>
<tr>
<th>Category</th>
<th>Frequency</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Problem Comprehension</td>
<td>3</td>
<td>14.29%</td>
</tr>
<tr>
<td>Decomposition</td>
<td>14</td>
<td>66.67%</td>
</tr>
<tr>
<td>Structuring</td>
<td>3</td>
<td>14.29%</td>
</tr>
<tr>
<td>Mental Simulation</td>
<td>1</td>
<td>4.76%</td>
</tr>
<tr>
<td>Evaluation</td>
<td>0</td>
<td>0.00%</td>
</tr>
</tbody>
</table>
David designed the system based entirely on his experience from a previous database class. What he did for this design session was to create two database tables, one for the patron and the other for the librarian. The rest of his cognitive activities were almost solely describing what attributes those tables should have and why. As a result, almost sixty percent of his episodes (60.87%) were decomposition (as shown in Table 4.12). Initially he indicated his design was done in episode number eleven. He then continued and finished the rest of the episodes after being asked by the experimenter to explain how the tables could be manipulated to accomplish library functions that perspective users would see on a computer screen. 

By analyzing David’s cognitive episodes (assuming he was earnest), it is difficult to tell what mental model he created for this design. It seemed like he simply copied what he did for his previous project, which was similar to the problem conceptually.

Basically it’s similar to what I designed in database…This is what I did to design the online bookstore and basically it’s similar to this concept.

Therefore the whole process was just the design of a database to support the system he was asked to design. Although he missed the meaning and goal of the design scenario, whether it was due to his overlooking the intent or the blunt problem description, one could see that his mental model, or approach to generate a mental model, relied heavily on prior experience. Prior experience played a crucial role in his problem comprehension, which is defined as the stage
when a participant forms his mental model for the problem description. Because his previous
database design experience for an online bookstore did not require functional implementation that
manipulated and processed data, he skipped all of the functional tasks of the system, such as the
looking up, checking in, and checking out of books. He mentioned those features as if they were
unrelated to the problem. All he focused on was the database design for the online library
management system.

**Design Session II:**

Table 4.13

Cognitive Process Frequency by David in Design Session II

<table>
<thead>
<tr>
<th>Category</th>
<th>Frequency</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Problem Comprehension</td>
<td>15</td>
<td>34.09%</td>
</tr>
<tr>
<td>Decomposition</td>
<td>20</td>
<td>45.45%</td>
</tr>
<tr>
<td>Structuring</td>
<td>8</td>
<td>18.18%</td>
</tr>
<tr>
<td>Mental Simulation</td>
<td>1</td>
<td>2.27%</td>
</tr>
<tr>
<td>Evaluation</td>
<td>0</td>
<td>0.00%</td>
</tr>
</tbody>
</table>

Figure 4.21 A scatter plot representing David’s chronological design processes in design session II.
In the beginning of design session two, David said “I need two classes for patron and librarian.” It is clear that he was trying to incorporate what he had learned in IST240 class into the current design. However, there were several breakdowns during the design process.

Let's say if the patron looking for, like the IST book, so let’s say here is IST. And this is another’s book, and … “Others”, um, the books that’s not match and it’s, it shows in that, um… Like this is just for another book. [In describing search function]

First, from his description, he divided all the collections of the library into two categories after search criteria were specified: books that matched the criteria and books that did not, rather than extracting books that matched the search criteria. This seems to violate the concept of a search function. Second, being able to see a library only from a patron’s perspective hindered his design ability. When the experimenter asked about an add function for the librarian to add a new library item to the system. He replied, “Librarian has to, are able add to the library collection? But the book is not stored in the system yet!”

Obviously, his memory stuck at the patron’s search function. David eventually said that if a library item was not in the current library collection, then it was in the category of “others”. The problem comprehension on scatter plot, Figure 4.21, demonstrates that David started the session by approaching the problem from the patron’s perspective. When being asked about functions for a librarian, he struggled to see that part of the system. As a result, there were several problem comprehension episodes which appeared after the midpoint of the design process, when normally the participants should already have their own mental models created.

**Design III:**

David started the third design session with:

This bookstore has two, um, three functions, for, one is for patron, and the other one librarian, and the third one is browser because the user needs to browse the
book, and, um, to view the book what they are looking for. So I, I assume that I need browser as, as another function in this bookstore system.

There were two noteworthy ideas in this statement. First of all, the instruction used “an online library management system” rather than “a bookstore”. In fact, there was no mention in the instructions of a bookstore. Some functions or tasks overlapped with each other, but differences between a library and a bookstore cannot and should not be ignored. For example, normally a bookstore does not loan books to its customers, so there is no need for a renewal function. In this regard, David probably used his previous project as a template to understand this scenario and generate a mental model identical to what he had made for a bookstore management system. Or he simply might have recalled his previous model and substituted it as response to these requirements. Secondly, a browser is a client program and it does not accomplish any functionality. David, however, defined it as another “function” of the library system. He also stated that it allowed users to look up library items, which was not correct. A browser only presents the results of a look up function, not fulfilling the function in actuality. In fact, the browser was only associated with looking up and checking out library items in his design process. I believe that David probably had a mental model that replaced commonly used UI functions with the browser.

And browser, um, allow you to look up for the authors, or the titles of the book…but this class is within the browser once you go online, then you can check out, you can search for the book you are looking for.

Table 4.14
Cognitive Process Frequency by David in Design Session III

<table>
<thead>
<tr>
<th>Category</th>
<th>Frequency</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Problem Comprehension</td>
<td>5</td>
<td>21.74%</td>
</tr>
<tr>
<td>Decomposition</td>
<td>10</td>
<td>43.48%</td>
</tr>
<tr>
<td>Structuring</td>
<td>7</td>
<td>30.43%</td>
</tr>
<tr>
<td>Mental Simulation</td>
<td>1</td>
<td>4.35%</td>
</tr>
</tbody>
</table>
In this design session, the combination of problem comprehension, decomposition, and structuring accounted for more than eighty percent (84.61%) of the total episodes as illustrated in Table 4.14. There was only one episode of mental simulation and no evaluation. Normally a UI-based mental model would generate more mental simulation than David demonstrated. The lack of mental simulation points to David’s inability to construct a feasible mental model. Even though David created a mental model based on how users will behave in the system, the model was not workable. Therefore he could not simulate some functions.

Figure 4.22 A scatter plot representing David’s chronological design processes in design session III. Figure 4.22 illustrates his decomposition and structure of behavior for the library system. However, there is no evidence of a systematic approach in this design.

To summarize, David did not have a thorough understanding of the library functions, nor did he form a solid mental model that could be executed. What he saw as the system was still no deeper than what people would see from the user interface.
Summary

Figure 4.23 Comparing David’s three design sessions by the number of design-related episodes.

Figure 4.24 Comparing David’s three design sessions by the percentage of design-related episodes.

Figure 4.23 and 4.24 illustrate the lack of mental simulation and evaluation in all three sessions. It is clear that David could not construct a robust mental model by the end of the study. Although he used decomposition episodes, his inability to design came from either problem comprehension, decomposition, or structuring.

I kind of get the idea but I just don’t know how to put together. (In design session two)

I just don’t know how to draw this…. But I get the idea what you said. (Also in design session two)
Even though he claimed that he understood the system, his verbal protocol did not support his claim. He struggled to understand the design scenario and was unable to map from the instructions to his own design. He also had problems with creating new components to assist him in this design. What he showed on design documents was largely based on the instruction sheet. For him, designing a system he had no experience with was challenging and he had difficulty stepping away from his previous database design experience. There was no observable improvement during the semester in terms of software design. He had no evaluation episodes in any of the three sessions and the number of mental simulation episodes was universally low.

David did try to use object-oriented concepts in his design in the beginning of the third design session, which he did not attempt in the first or second session. The programming knowledge enabled him to design software systematically at a lower level. It did not improve his higher level design knowledge, however. It was not apparent that he would improve this higher level design knowledge when he had built a robust foundation with his lower level design knowledge. Nevertheless, the higher level design knowledge does rely on the support from lower level ones. Therefore, it will be a question to be answered in the future: whether David can improve his mental simulation and evaluation ability through consolidating his problem comprehension, decomposition, and structuring abilities.

Lazarus

Demographic Data:

Lazarus, who was 20 years old when participating in this study, was an IST major third-year undergraduate student. He was a Computer Science and Engineering major before he transferred to IST. He had about 10 years of experience using computers for checking emails,
doing homework, playing video games, chatting with friends, and browsing the Internet. He also
had experience using C++, APS.NET, C#, HTML, and PHP. He took the IST210 course on
database design, and created a web application as the final project using PHP and SQL Server
2005. In terms of being involved in a software design project, he was in a group that did a Tetris
game using C++ for CSE121: Introduction to programming techniques.

**Design Session I:**

After reading the instructions, Lazarus began his design with two major UIs, one for the
patron and the other for the librarian. Tasks for both users were then placed on the UIs. Table
4.15 illustrates that Lazarus is balanced on attending to cognitive processes in decomposition,
structuring, and mental simulation. He did problem comprehension and decomposition early, and
then he started structuring and mental simulation. The analysis of his verbal protocol also
indicates that he was very focused on implementing the tasks for both user groups. Relying on UI
as his mental model, drew his attention into the details of each task early and forced him to
recreate part of this mental model as Figure 4.25 indicates. Half (four episodes) of his problem
comprehension occurred late in the session.

Table 4.15

<table>
<thead>
<tr>
<th>Category</th>
<th>Frequency</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Problem Comprehension</td>
<td>8</td>
<td>18.60%</td>
</tr>
<tr>
<td>Decomposition</td>
<td>12</td>
<td>27.91%</td>
</tr>
<tr>
<td>Structuring</td>
<td>11</td>
<td>25.58%</td>
</tr>
<tr>
<td>Mental Simulation</td>
<td>12</td>
<td>27.91%</td>
</tr>
<tr>
<td>Evaluation</td>
<td>0</td>
<td>0.00%</td>
</tr>
</tbody>
</table>
Also in this design session, Lazarus did not use any specific programming knowledge in his design. The design document consisted of texts, blocks, and arrows, which depicted how he understood the system. He even reminded himself to think of the problem in design, not in implementation.

But I’m not thinking about like in terms of programming or other things. I just want to [be] clear [for] myself about the problem.

His programming experience might have enabled him to concentrate on the structure and not the implementation.

**Design Session II:**

Lazarus switched from a UI-based mental model to a programming-based model that used object-oriented concepts to design the system in this session. The implication of this change was the occurrence of problem comprehension, which all occurred early in the design process (Figure 4.26). Another implication was the dramatic reduction of mental simulation processes as showed in Table 4.16. Lazarus created three major classes: book, patron, and librarian. He spent all but one episode designing for book and patron. The last episode of his design was to create a librarian class and its attributes and methods. He created the book class and its attributes first.
Methods used to manipulate books were created right after the class was created. Patron class attributes and methods were created after book class was done.

Table 4.16

Cognitive Process Frequency by Lazarus Design Session II

<table>
<thead>
<tr>
<th>Category</th>
<th>Frequency</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Problem Comprehension</td>
<td>4</td>
<td>10.81%</td>
</tr>
<tr>
<td>Decomposition</td>
<td>16</td>
<td>42.24%</td>
</tr>
<tr>
<td>Structuring</td>
<td>16</td>
<td>42.24%</td>
</tr>
<tr>
<td>Mental Simulation</td>
<td>1</td>
<td>2.70%</td>
</tr>
<tr>
<td>Evaluation</td>
<td>0</td>
<td>0.00%</td>
</tr>
</tbody>
</table>

Figure 4.26 A scatter plot representing Lazarus’s chronological design processes in design session II.

In episode 11 he asked “what does check-in mean?” that did not occur in design session one. In design session one, Lazarus did not attempt to examine what the “checkin” function would do to the user because he used a UI-based mental model where he could simply use a button to represent that task. In this design session, however, he used a mental model that was based on programming language, and he needed to know the procedural implications of this task. Using different guidance to construct a mental model, therefore, has profound implications on what the structure of design will look like and what types of cognitive processes will be required.
Design Session III:

Lazarus used a similar design approach to session two. Again, he started his design by creating two major classes: the patron and the librarian. The results of these design patterns were very similar to design session two. Because he used a programming-based mental model again, he needed to know the meaning of a specific task in order to understand its procedure. In design session three, he asked, “Um, does check-in mean like when I return a book, the librarian checks in the item?” and “Um, what does it mean by send notifications?” It was clear that the use of this type of mental model required Lazarus to know the details of a task in order to use programming symbols and concepts to represent his design. Although it was unlikely, Lazarus might not have been familiar with the terminologies used in the instructional brief or with the domain knowledge, which was the function of a library.

It seemed that using a programming-based mental model also caused the designer to concentrate on problem comprehension and structuring as is shown in Table 4.17. Figure 4.27 also shows that Lazarus systematically used a cognitive process sequence of decomposition, structuring, and mental simulation. Unlike in design session two where all problem comprehension episodes occurred early, the problem comprehension episodes in design session three were spread out due to the inquiries about the meaning of certain tasks, and some were because he had to modify the existing design and re-read the instructions to accommodate a new function.

Table 4.17
Cognitive Process Frequency by Lazarus Design Session III

<table>
<thead>
<tr>
<th>Category</th>
<th>Frequency</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Problem Comprehension</td>
<td>7</td>
<td>9.59%</td>
</tr>
<tr>
<td>Decomposition</td>
<td>29</td>
<td>39.73%</td>
</tr>
<tr>
<td>Structuring</td>
<td>24</td>
<td>32.88%</td>
</tr>
<tr>
<td>Mental Simulation</td>
<td>11</td>
<td>15.07%</td>
</tr>
</tbody>
</table>
Figure 4.27 A scatter plot representing Lazarus’s chronological design processes in design session III.

Summary:

Lazarus demonstrated two very different types of design mental models. One was a UI-based model in session one, and the other was a programming-based mental model in sessions two and three. The UI-based model allowed him to bypass some of the functional details of a task, but it relied on the use of mental simulation to imitate users’ behaviors. Figures 4.28 and 4.29 illustrate that mental simulation occurred more frequently, in terms of both frequency and percentage in design session one. When using a programming-based model, however, the details are pivotal to a design, and designers use a pseudo-code format to assemble the system. Note that in design session two, there were very few mental simulations, while he did several mental simulations episodes in session three. It is possible that he did not feel comfortable about his design because he had only studied Java programming for about five weeks, and consequently, did not want to review his own design; whereas, in design session three, he used mental simulation to review his design. However, there is no evidence to confirm or disapprove this conjecture. It might just be that he chose not to in session two, and opted for self-reviewing in session three.
Mary

Demographic Data:

Mary, who was 21 years old when participating in this study, was an IST major, third-year undergraduate student. She had about 13 years of experience using computers for checking emails, checking blogs, doing school work and research, and a little programming and scripting. She had experience using C++ as part of the IST curriculum. In IST210 course, where she learned
database design and SQL, she did a group final project on a music database. That final project was to demonstrate their SQL language competency, therefore, no web scripts or interfaces were involved.

**Design Session I:**

Mary started the design session by separating the system into two categories, the patron and the librarian, and she listed all tasks under each category respectively. She read through the instructions to generate the list and categorized each task. She generated a visual representation of this list, which had vertical lines connecting from the librarian or the patron to a block representing the database to help her during the design process. Each vertical line represented a task that either the librarian or the patron could perform. From this drawing, she seemed to focus the interaction between either the patron or the librarian and the database. There was no interaction between the librarian and the patron, although some tasks may have involved both the patron and the librarian at first glance (Figure 4.30).
Figure 4.30 Mary’s first visual representation of the online library management system in design session I.

When that diagram was finished, she decided to draw a picture of a stick figure standing in front of a computer so she could envision a scenario which a person was using a library computer.

Sometimes even stick figures help. So I have, I’m drawing a stick figure this helps just seeing them in my...I can’t draw at all.

After the drawing was done, she, again, separated users, who logged into the system, into two categories, which were not surprisingly, the patron and the librarian. Tasks were being filled into different branches that represented the users’ choice of task.

So if I have a user who is logging on to OLMS or the GUI for the library management system, they’re either logging into patron or librarian. And after they’ve logged in, they are going to decide what they want to do.

The majority of her cognitive activities were problem comprehension, as shown in table 4.18, as she did not get into the details of the tasks. Her mental model was dominated by what users would do, i.e., a UI-based mental model. There are significant mental simulation episodes, comparable to other design activities. Figure 4.31 shows that most of the structuring and mental
simulation occurred in the phase when she was using a stick figure to trace users’ behavior. Otherwise, most of the cognitive processes were problem comprehension.

Table 4.18

Cognitive Process Frequency by Mary Design Session I

<table>
<thead>
<tr>
<th>Category</th>
<th>Frequency</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Problem Comprehension</td>
<td>22</td>
<td>45.24%</td>
</tr>
<tr>
<td>Decomposition</td>
<td>6</td>
<td>14.29%</td>
</tr>
<tr>
<td>Structuring</td>
<td>10</td>
<td>23.81%</td>
</tr>
<tr>
<td>Mental Simulation</td>
<td>7</td>
<td>16.67%</td>
</tr>
<tr>
<td>Evaluation</td>
<td>0</td>
<td>0.00%</td>
</tr>
</tbody>
</table>

Mary did state that she did not have design experience and her software design problems were usually less descriptive. As a software design novice, the UI-based mental model fits her competency appropriately because she could concentrate on the organization of those tasks without worrying about the details.

**Design Session II:**

In this session, Mary changed her mental model from a UI-based to a programming-based one. She tried to incorporate object-oriented concepts such as classes, attributes, and methods into her design. She started the design by creating two classes, the patron and the librarian. Then she
wrote a list of tasks that needed to be done by them and moved them to the classes: methods by looking for verbs, attributes by looking for nouns. The final step was connecting the objects with a database by those methods.

Table 4.19 shows that, in terms of the number of episodes, most of the cognitive activities she attended to in this design session were problem comprehension, decomposition, and structuring, as the programming-based mental model dictated a more detail-oriented approach. Figure 4.32 shows her design behavior related to understanding the problem and breaking problems down before connecting some sub-components with others.

Table 4.19
Cognitive Process Frequency by Mary Design Session II

<table>
<thead>
<tr>
<th>Category</th>
<th>Frequency</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Problem Comprehension</td>
<td>16</td>
<td>39.02%</td>
</tr>
<tr>
<td>Decomposition</td>
<td>14</td>
<td>34.15%</td>
</tr>
<tr>
<td>Structuring</td>
<td>11</td>
<td>26.83%</td>
</tr>
<tr>
<td>Mental Simulation</td>
<td>0</td>
<td>0.00%</td>
</tr>
<tr>
<td>Evaluation</td>
<td>0</td>
<td>0.00%</td>
</tr>
</tbody>
</table>

Figure 4.32 A scatter plot representing Mary’s chronological design processes in design session II.

There was a noteworthy non-design episode in which Mary mentioned that she had trouble understanding object-oriented programming. However, that did not prevent her from using a programming-based mental model. She seemed to follow specific instructions on how and where to look for clues for design.
I am looking at attributes, um, I am thinking library item, oh these are all methods, these are all I mean, and I know they are methods because I see that these are verbs…I am looking at the nouns which to me indicate the attributes…I am underlining the verbs, which should be the methods.

**Design Session III:**

<table>
<thead>
<tr>
<th>Category</th>
<th>Frequency</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Problem Comprehension</td>
<td>22</td>
<td>29.33%</td>
</tr>
<tr>
<td>Decomposition</td>
<td>33</td>
<td>44.00%</td>
</tr>
<tr>
<td>Structuring</td>
<td>12</td>
<td>16.00%</td>
</tr>
<tr>
<td>Mental Simulation</td>
<td>4</td>
<td>5.33%</td>
</tr>
<tr>
<td>Evaluation</td>
<td>4</td>
<td>5.33%</td>
</tr>
</tbody>
</table>

In this design session, Mary generated a lot of problem comprehension episodes (37.5%; Table 4.20). The main reason for this was that she did not create a representing system from her own mental model. Instead she relied on the instructions to be her guide. The pitfall of using the instructions rather than her own mental model or external model, was that the instructions were not organized according to her understanding of how the system functioned or her knowledge base. This created some breakdowns in her design later when it became large and complex. She had to keep referring back to the instructions and struggling with the terminology that had conflicted with her comprehension.

…**basic information**, by **basic information** I think it means information about the books.

But by **record**, I think I really mean **history**, that’s more like…

…add, remove, and edit items, did I include that yet?
Without creating a mental model in her memory or an external model on paper, she missed a lot of the links that were not displayed in the textual description, and obviously she was not able to store the whole mental model in her STM.

![Figure 4.33 A scatter plot representing Mary's chronological design processes in design session III.](image)

However, she did demonstrate consistent design behavior (Figure 4.33), decomposition followed by structuring, and then mental simulation. She also demonstrated the ability to evaluate multiple design options and then select one that was better and more flexible, which she acknowledged herself:

Yeah, I think that’s better to do this. That way it’s more, it’s broader. [On debating whether to create a class of book or item]

One thing she apparently learned from class was to look for nouns as indicators of creating attributes and verbs as indicators of creating methods in the design description. She followed that principle strictly.

In the end, she did not review or test whether her design worked as expected. This was not an indication of missing mental simulation ability. It might be the confidence that she had done everything that was required for the scenario. In conclusion, she had a mental model that replicated what was written in the instructions. Therefore her mental model had more static components and fewer relationships and interactions. Her design artifact could attest to this conclusion because of its numbered tasks and lack of arrows pointing to other tasks.
Summary

Mary used a UI-based mental model in the first session and a programming-based mental model in the second and third sessions. Figure 4.34 shows that the increase of decomposition is the most dramatic and obvious change. As discussed in other participants’ summaries, a programming-based mental model will tax more low-level cognitive processes, such as problem comprehension and decomposition. This phenomenon is depicted in Figure 4.34 and 4.35 where the number of decomposition episodes has been growing steadily.

![Figure 4.34 Comparing Mary’s three design sessions by the number of design-related episodes.](image1)

![Figure 4.35 Comparing Mary’s three design sessions by the percentage of design-related episodes.](image2)

In terms of the occurrence of Mary’s mental simulation, the UI-based mental model generated more mental simulation than the programming-based mental model. Furthermore, there
was no mental simulation in session two, which was the first time she had demonstrated using a programming-based mental model. Although she did not mention the reason, it was possible that it was because she had been in Java class for less than two months. To simulate the system using a programming-based mental model would require executing pseudo-code mentally, which might have been too challenging for Mary as a novice programmer. In session three, she did demonstrate four mental simulation activities.

Matthew

**Demographic Data:**

Matthew, who was 24 years old when participating in this study, was an IST major undergraduate student, and in landscape contracting business. It was his seventh year at a university. He enrolled in landscape architecture and then transferred to the IST program. He was in the second year in the IST. He had 16 years of experience using computers for checking emails, doing school work, browsing the Internet for leisure and for learning computer concepts. He had one semester of C++ experience with regard to computer programming.

**Design Session I:**

Table 4.21

<table>
<thead>
<tr>
<th>Category</th>
<th>Frequency</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Problem Comprehension</td>
<td>12</td>
<td>19.67%</td>
</tr>
<tr>
<td>Decomposition</td>
<td>27</td>
<td>44.26%</td>
</tr>
<tr>
<td>Structuring</td>
<td>19</td>
<td>31.15%</td>
</tr>
</tbody>
</table>
In the first design session, Matthew started with a lot of decomposition processes, as indicated in Table 4.21. He read the instructions and created his own “to-do” list for reference. There were several occasions, according to his verbal data, that he was using this new representation as a design aid.

I’m putting the list together so I can do a comparison in a way that helps me be able to think.

I’m making this list so I know what to do later.

I just crossed that out cause I made a mistake. I’m not erasing it because I like to know where I screwed up next time.

The fact that he could create the representing system without any problem, indicated that he had sufficient skills for comprehending the system and organizing that knowledge. He then started to design individual components based on the mental model he had just created. The design procedures were filled with decomposition processes, interwoven with problem comprehension processes and structuring processes. It was common for him to create a sub-component and then find out that something needed to be changed. One change would trigger another, either in terms the relationship between components or of the component itself. This
pattern lasted until the end of the design. Very few mental simulations (one episode) or evaluation (two episodes) were observed during the design process. The evaluation activities were conjectures about some improvement that could be done, but that he did not have the knowledge to accomplish, according to Matthew’s verbal data. The frequent changes to his representation of the system suggest that the structure of Matthew’s mental model was consistently changing during the design process. Figure 4.36 also illustrates Matthew’s opportunistic design behavior where decomposition and structuring activities are dispersed throughout the design process. Although he could create a representing system in a short time, the representing system was incomplete or incorrect. In the end, when he felt that he fulfilled everything on the list, he stopped working on the design.

**Design Session II:**

In this session, Matthew started the design process by creating a log-in page, which branched out to the worker (librarian), who had more options, and the client (patron), who had fewer options. His mental model was similar to a UI-based one, which meant his cognitive process walked through actions on the UIs. As a result, he had to modify his design often, either adding new features or making connections between the different tasks, functions, and pages.

…from here, you can also proceed to the screen which would loop back to a new search, have a link to the personal page or, um, on the search could also a link to the personal page…

| Table 4.22 |
|---|---|---|
| Cognitive Process Frequency by Matthew in Design Session II | | |
| Category | Frequency | Percentage |
| Problem Comprehension | 3 | 6.67% |
| Decomposition | 17 | 37.78% |
| Structuring | 12 | 26.67% |
Table 4.22 shows that 93.34% of the design-related episodes are decomposition, structuring, and mental simulation processes. Although these three categories are fairly balanced, there is no design pattern. In addition, Figure 4.37 shows that episodes from all three categories are dispersed evenly. In this design, he demonstrated the opportunistic design pattern again. There was no systematic procedure such as top-down or bottom-up types of methodology. Matthew often found a new feature or missing attribute when he was designing another seemingly unrelated task. Consequently, he would forget what he was attending to previously. Here is one example: “This one back to here was, um…what was it? Um…oh, notification.”

**Design Session III:**

In the third session, Matthew started by listing tasks under two categories, the patron and the librarian, and followed that by creating his first page, a log-in screen. There was, however, no screen or user interface design at all. Rather, Matthew wrote down tasks related to the log-in screen and how those tasks should be done in text format. Using the same format, he created nine categories: log-in screen, patron screen/librarian screen, catalog database, patron database,
reserve/renew function, fines function, online resources, history page, and resource. Although it said “screen” in some categories, they were, in fact, tasks for those categories. Also, these categories were created one by one, but their respective descriptions were not. Descriptions for each category included who can use it, what attributes it should have, the tasks that belonged to it, and what output it might have. Matthew, again, used an opportunistic approach in this design, as illustrated by the fact that he would realize that something was missing and change it accordingly.

Due to the behavior of putting functions and attributes into different categories, and the style of the design document organization, the most frequent cognitive process was decomposition as showed in Table 4.23. All other cognitive processes were scarce. Furthermore, decomposition activities were dispersed across almost the whole process until, toward the end, where there were some structuring and mental simulation activities (Figure 4.38). Four episodes, coded as non-design, were similar to the following:

Let’s see, just look over the main sheet, look at that again, and see if there is anything that jogs my memory for any other needed information.

There were several possibilities of what cognitive activity he was attending to at that time. Without further information and evidence, however, there was no way to establish an appropriate code for such episodes.

Table 4.23

Cognitive Process Frequency by Matthew in Design Session III

<table>
<thead>
<tr>
<th>Category</th>
<th>Frequency</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Problem Comprehension</td>
<td>7</td>
<td>12.28%</td>
</tr>
<tr>
<td>Decomposition</td>
<td>26</td>
<td>45.61%</td>
</tr>
<tr>
<td>Structuring</td>
<td>7</td>
<td>12.28%</td>
</tr>
<tr>
<td>Mental Simulation</td>
<td>3</td>
<td>5.26%</td>
</tr>
<tr>
<td>Evaluation</td>
<td>0</td>
<td>0.00%</td>
</tr>
</tbody>
</table>
Figure 4.38 A scatter plot representing Matthew’s chronological design processes in design session III.

**Summary**

Analyzing Matthew’s verbal protocols from all three sessions indicates that he does not seem to follow any specific method of design. In the first session, he used half diagram and half text information to create his design document. In the second session, he used a UI-based mental model to create this design document. In the third session, he used only text descriptions to describe his design. Although he always starts from a log-in page, what follows is very different. There is no evidence that learning Java programming for one semester changed his design behavior.
In terms of each type of cognitive process, figures 4.39 and 4.40 show that the most obvious difference is the mental simulation in design session two. In that session, he used a UI-based mental model to design the system, which may have caused him to discover different functions and tasks that he got from a scenario walk-through approach. Therefore, there were an abnormally high number of mental simulation activities in design session two.
Zeke

Demographic Data:

Zeke, who was 20 years old participating in this study, was an IST major third-year undergraduate student. He had 10 years of experience using computers for communication, surfing the Internet, instant messaging, and doing class work. His only programming experience was using C++ as part of the IST curriculum and taking the current Java programming course.

Design Session I:

Table 4.24
Cognitive Process Frequency by Zeke in Design Session I

<table>
<thead>
<tr>
<th>Category</th>
<th>Frequency</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Problem Comprehension</td>
<td>0</td>
<td>0.00%</td>
</tr>
<tr>
<td>Decomposition</td>
<td>16</td>
<td>57.14%</td>
</tr>
<tr>
<td>Structuring</td>
<td>6</td>
<td>21.43%</td>
</tr>
<tr>
<td>Mental Simulation</td>
<td>6</td>
<td>21.43%</td>
</tr>
<tr>
<td>Evaluation</td>
<td>0</td>
<td>0.00%</td>
</tr>
</tbody>
</table>

Figure 4.41 A scatter plot representing Zeke’s chronological design processes in design session I.
The thing that stood out during the verbal analysis after Zeke’s first design was that there was no problem comprehension process at all. Although Zeke did read the instructions quickly, he started right into decomposition and other processes, with decomposition being the dominant one (Table 4.24). We could assume that Zeke had a mental model because he did not refer to the instructions during the design. In addition, this mental model seemed to be a mix of his perception of a library and the description of the scenario. The lack of written representation of the system hampered his design during the process in the sense that some key components were missing from the final design document. It also forced him to do a lot of modification throughout the design process.

So how did he handle the design task? The analysis indicates that he leaned heavily on simulating the transactions between patrons and librarians during the design task. In ending the design task, he said “we need something that would allow the student to check-out, reserve, or renew a book. I think that’s about it [for the design].” There was not any detail about how programmers would achieve those tasks, nor did he try to investigate the interrelationships between those components. This statement was made in episode thirteen. Most of the verbal data was vague and abstract, especially on the functional level. Even after being asked to explain in detail about how the components worked to achieve the ultimate design goal, Zeke could only come up with very preliminary design descriptions.

Zeke’s first attempt can be summarized as a need-based, transaction-driven design. There was no complex cognitive process, judging by the quality of his verbal protocol. The decomposition, structuring, and mental simulation processes observed were not as complicated as what the experts did. The decomposition processes could be easily found in the instructions such as patrons needing to be able to search for library items, review and change personal information, access their library records, etc. Mental simulation processes were simple descriptions rather than the manipulation of components and relationships. For example, Zeke mentioned:
...we’d have an option in the program that would allow the librarian to e-mail a student if a book was overdue and tell them you know they have x number of days to get it back of they’ll be fined or their bursar account would be fined. Something like that.

We need a program that would again read the, the strip on the back of the card and would then query, would take this into a program, figure out the student’s name, student ID number and then would send that back to the central server and query that central server.

It might be that Zeke’s lack of programming experience hindered his ability to manipulate the design scenario in a way that could be implemented by programmers.

Design Session II:

Zeke started the second design session by listing all the tasks after he read the instructions. He then drew a circle to represent a database in the center of the design document. On top of the page, he created a patron-user interface and started to fit tasks into the interface one by one. The main patron page was also connected to two other pop-up windows to display additional information. All tasks were seen as queries to the database.

...we do book look-up and, you know, like a search button, that’s basically we are going to query the database and we can do the book look-up...this is pretty much just different query...this will all be querying this database.

Zeke created a librarian page on the bottom of the design document using the same design pattern, explaining each task from the task list, based on the concept of database query, and creating connections as needed. The intention of asking the designer to make their design less redundant on the provided design scenario (appendix B) was to elicit their ability to identify similar tasks and combine several tasks. Because Zeke used a UI-based mental model, his understanding of reducing redundancy was making the UI more efficient by displaying relevant information on the same page.
…to keep things flowing, and keeping them less redundant. We can also display what books they have reserve or renew and records of books they rented or borrowed in the past on this page. So reserve…renew…records…

Table 4.25

Cognitive Process Frequency by Zeke in Design Session II

<table>
<thead>
<tr>
<th>Category</th>
<th>Frequency</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Problem Comprehension</td>
<td>10</td>
<td>31.25%</td>
</tr>
<tr>
<td>Decomposition</td>
<td>8</td>
<td>25.00%</td>
</tr>
<tr>
<td>Structuring</td>
<td>6</td>
<td>18.75%</td>
</tr>
<tr>
<td>Mental Simulation</td>
<td>8</td>
<td>25.00%</td>
</tr>
<tr>
<td>Evaluation</td>
<td>0</td>
<td>0.00%</td>
</tr>
</tbody>
</table>

Figure 4.42 A scatter plot representing Zeke’s chronological design processes in design session II.

Table 4.25 shows that the distribution of cognitive processes is balanced among problem comprehension, decomposition, structuring, and mental simulation. There is no evaluation however. Figure 4.42 shows that many problem comprehension processes occurred in the beginning when he created a task list. Decomposition, structuring, and mental simulation are interwoven in a pattern where decomposition occurs earlier, and mental simulation occurs later.
**Design Session III:**

In this design session, Zeke used almost the same strategy, except that he did not create a task list this time. Again, he created a database in the center of the paper, then, he designed a patron page and fitted tasks into the patron page. He created a librarian page afterwards. In this design session, however, he created more pages for both the patron and the librarian. Both the patron pages and the librarian pages were revolving around the database. Learning Java programming in class did not seem to affect his design processes and mental model.

Table 4.26 shows that design processes were balanced between problem comprehension, decomposition, structuring, and mental simulation, similar to design session two. There was no evaluation.

### Table 4.26

<table>
<thead>
<tr>
<th>Category</th>
<th>Frequency</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Problem Comprehension</td>
<td>7</td>
<td>21.21%</td>
</tr>
<tr>
<td>Decomposition</td>
<td>9</td>
<td>27.27%</td>
</tr>
<tr>
<td>Structuring</td>
<td>7</td>
<td>21.21%</td>
</tr>
<tr>
<td>Mental Simulation</td>
<td>10</td>
<td>30.30%</td>
</tr>
<tr>
<td>Evaluation</td>
<td>0</td>
<td>0.00%</td>
</tr>
</tbody>
</table>

Figure 4.43 A scatter plot representing Zeke’s chronological design processes in design session III.
Figure 4.43 shows that the problem comprehension, decomposition, and structuring were spread out across the design process. Mental simulation seemed to condense into small groups. That was because Zeke’s design was influenced by a UI-based mental model and simulation often happened after a page had been constructed.

**Summary**

The analysis of Zeke’s verbal protocol indicates that his mental models were very similar for the three sessions. He used almost identical approaches to his design and the processes are also similar. According to Figure 4.44 and 4.45, the biggest difference was that there was no problem comprehension and the decomposition episode took up more than 50% of the total episodes in session one. This was due to his design behavior in that he did not create a task list to represent the system. It suggests that his first design was based on what he had in his memory in the first session.

If we examine more carefully the quality of those cognitive processes, it is not hard to find that his design tactics are centered on arranging and categorizing the tasks. The decomposition process is more related to entity decomposition, the structuring process is more about data flow structuring, and the mental simulation is more about behavior simulation. The use of a UI-based design mental model also made his design process opportunistic rather than systematic.

Let’s see, look up overdue items, kind of jumping around here, go back to this item here, um, main library page. (In design session three)
The missing evaluation episodes for all three sessions can be seen as an indicator that he is just following the instructions and reorganizing those tasks. Because those tasks are visible for a person who is familiar with the library system, it is not difficult to rearrange those tasks into one design document.

**Discussion**

In this section, I will discuss briefly the research questions and how the verbal analysis informs us on those issues.
Question 1: **What are the characteristics of the novice programmer’s software design mental model?** The verbal analysis of novice programmers suggests that novice programmers have some shared characteristics. First, the creation of a mental model relies heavily on prior experience and knowledge. When novices need to solve a problem that exceeds what they already know, drawing related knowledge from memory becomes a logical choice to solve the problem (Driscoll, 2000, p. 195; Rumelhart & Norman, 1978), especially when the discrepancy seems small. In the first design session, every participant used the database design concept from their prior coursework to solve the problem. In David’s case, he replaced the library system with the online bookstore he designed in the database design course. Some also adapted their programming knowledge and concepts into creating their mental model, resulting in a complete change to their design behavior. With the limited resources they have, novice programmers had to find prior knowledge that is compatible with their current situation. The experience of interacting with similar systems is also activated in order to assist them, whether the experience is partial or complete. So, when facing problems, they cannot use a thoughtful and logical way to come up with a solution. Instead, a template from memory is what a novice programmer will do first.

Second, novice programmers’ mental models are inadequate and incomplete (Kessler & Anderson, 1989), either because they do not have sufficient ability to generate multiple views of a system, or they do not use this strategy. A complex system or problem usually consists of subsystems. Since they are subsystems, they have their own properties and ways of interaction. Solving these subsystems, therefore, can be independent. Novice programmers seem to be restricted to a single mental model.

Question 2: **What type of mental model does a novice programmer use?** Novice programmers in this study clearly use two types of mental models. A UI-based mental model was common among all participants because it is probably the easiest way to convert the design scenario to individualized representations. The process only requires prior experience with the
library system, which most college students already have. Every participant is capable of creating some mental model, whether it is complete or incomplete. Although everyone created a UI-based mental model, each person’s design documents are different. This variation comes from their perception of what is important, their experience with using libraries, and the libraries with which they interacted. The payment method for find, borrowing history, and usability of the interface are some aspects that were mentioned. As a result, the final design documents vary.

In design sessions two and three, some participants used a programming-based mental model which has properties and concepts that are inherited from programming languages, along with their detail-oriented nature. Using a programming-based mental model makes designers focus on lower level cognitive activities, such decomposition and structuring. Nevertheless, many participants will rely on the UI-based mental model while incorporating a programming-based model.

Question 3: *What causes novice programmer’s software design mental model to change? And what are the changes?* In this study, six of out eight novice programmers showed that their design mental models were similar to themselves in all the sessions throughout this semester, except for Lazarus and Mary who attempted to use a programming-based model. Even though the episodes of cognitive process varied in different design sessions, their mental models and approach did not change dramatically. So, the mental model does not change across sessions. On the other hand, although their mental models in different sessions are invariant, their mental models change consistently during the same design session. This is concluded from that fact that they have to change the structure of their designs or the properties of the entities. Two explanations can account for the frequent modifications. First, it may be that the details are missing from their initial mental model. During the design process, they discover more information and modify their mental models accordingly to accommodate the new information. Second, it may be that the novice programmers had a completely working mental model, but
because it was stored in their LTM, the details were forgotten during recall or too weak to be retrieved (Nelson, 1971) when created without rehearsal. Two novice programmers, Lazarus and Mary, changed their mental models after they learned Java programming for more than one month. Through analyzing their verbal protocol, we found that they both used very specific aspects of the problem instructions to help them in the design process. Being a novice programmer without too much cognitive process support, it is difficult to apply different methods in forming a mental model. With a set of very specific instructions, it’s easier for novices to use another type of mental model. Both Lazarus and Mary were able to use programming-based mental model in the second and third sessions. Had they failed in the second session, they might have fallen back to the previous type of mental model.
Chapter 5

CONCLUSIONS AND RECOMMENDATIONS

“Clearly, programming courses should teach methods of design and construction, and the selected examples should be such that a graduate development can be nicely demonstrated.” (Wirth, 1971, p. 221)

In this chapter, I will discuss the conclusions drawn from previous chapters and propose recommendations for future studies.

Conclusions

This study investigates, from the perspective of design, how novices form and use their mental models and what might cause their mental models to evolve, with the goal of making software engineering education more effective by understanding the learners.

Forming an adequate mental model is a critical factor for software design because the mental model controls designers’ behaviors and the cognitive processes that support those behaviors. According to the verbal analysis in this study, novices create individual mental models that govern their design behaviors and, in turn, their finished design products. Nevertheless, the structures and the contents of their mental models are simple and straightforward. First, let us look at the contents of their mental models. In the first design session, all tasks were taken at their face value from the instructions. For example, when the novices attended to the task of “editing a library item”, they either thought of it as a database query to databases or as a black box that would somehow finish the task. At first glance, it seemed novices’ limited experiences prevented them from procedural decomposition. However, experts used design tools such as the transition
diagrams and state diagrams to help them in the design process as well. If experts with extensive experiences and knowledge need to use external design tools and models, the process of procedural decomposition may be too complex in these circumstances for humans to retrieve all the related necessary components from LTM into STM at once (Miller, 1956). In order for novices to perform complex cognitive activities, learning appropriate design skills, such as decomposition and diagramming, would appear to be important.

Second, the structure of novices’ mental models is simple and linear. There is no complex structure observed from their verbal analysis and design documents. A typical structure in their solutions was to divide the system into two sub-components, one was the librarian and the other was the patron. After that, tasks were assigned into one or the other of these categories. The interactions between these two entities were examined only when necessary. For example, when a librarian checks out an item for a patron, the designer may want to record who checks out which item for whom. A basic problem-solving mental model for experts is organized so that it consists of large, more abstract elements on the higher level and more concrete, detailed elements on the lower level (Rumelhart & Norman, 1978). In the case of novices’ mental models, there were concrete elements in the higher level but no detailed representations in the lower level. Novices’ inability to generate detailed contents hindered their ability to create a more complete mental model. As a result, the relationships between elements were limited to observable actions among elements of the problem.

Two types of mental models emerged from this study. The first one was a UI-based mental model that was constructed by tracing user behaviors when they engage with UIs. A UI-based mental model is an easy one to construct because it requires only the experiences of interacting with a similar system. No special computer programming knowledge is required. Therefore, it is common for a novice programmer to use this approach to create a mental model. This mental model, however, suffers from functional isolation, which means the interaction
between tasks was minimal, because the novices often failed to see the underlying procedures and to make connections accordingly. Another type of mental model was programming-based, which decomposes an entity by its attributes and connects entities through functional relationships. While programming-based mental models inherit some properties from a UI-based mental model, they also utilize specific knowledge from a programming language. Two novices, Mary and Lazarus, switched their mental models from UI-based to programming-based, and the result was more a detail-oriented design with many procedural decompositions. The other six novices also showed their intentions of moving into a programming-based mental model, but with fewer details and an unsuccessful transfer.

Novices also reconstructed their mental models continually during the design process, more often than experts, perhaps due to the incompleteness of their initial mental models. As a result, novices attended to the processes of comprehending the problem and restructuring their mental models. Creating multiple mental models was not feasible because of the STM limitations. Throughout the design process, their mental models were monolithic and changing. On the other hand, experts created mental models to guide them through the design process, and both experts had multiple mental models, where some provided a global view of the system and others facilitated stepwise scripts in details. Markman and Gentner reported a similar observation that “people typically possess multiple models of complex systems, some highly context-bound and others more abstract.” (Markman & Gentner, 2001, p. 231)

Learning Java and object-oriented concepts provides another perspective to novices in the formation of mental models, and it reshapes their perceptions of problem representation, from behavior– and action–oriented, to function– and procedure–oriented. Unfortunately the effect of programming instruction does not appear to significantly change novices’ mental model formation, variety, and transformation in most cases.
Implications for Instruction

So how does this study help in terms of designing instruction? In a computer-programming classroom that requires a complex, multi-member project as part of the class, teaching should support students by using design tools to mitigate or offset the cognitive complexity of the design project. Learning computer programming itself is complex; if students cannot manage the additional cognitive complexity of software design, they will risk mental breakdowns, which even expert programmers experience, according to Guindon (1987). Traditionally computer programming prepares students with programming knowledge, but does not prepare them for making a robust mental representation of the problem. When dealing with simple and small problems, the power of a robust mental model may not be as important, but in the real world of research and design, the ability to handle the complexity of software development projects becomes extremely important.

Teaching mental model generation is not the same as providing everyone with the same problem representation. The instruction can be designed so that the students learn similar concepts and methods, and yet are able to create individualized mental models based on their own experiences. Such methods must also help students make their mental models external because in a complex problem, mental models will exceed the capacity of STM. Mapping a mental model into an external format will improve students’ problem solving performance, because the external model is easier to use as a retrieval cue (Beveridge & Parkins, 1987). It seems Mary and Lazarus both benefited from simple and clear instructions to follow by using simple techniques to map the instructions onto a mental model. Although the instructions they received were aimed at program generation, it helped the novices in creating programming-based mental models as well.
Implications for Cognitive Psychology

The formal study of the architecture of human cognition has been around for decades. A part of cognitive psychology research is to study mental representation for problem solving in different domains, of which software design is one. This study provides insight into how novice programmers construct and restructure their mental models when facing a complex software design problem. Two mental models related to software design were displayed and their implications on novices’ design behaviors were also described. Although there is still a long way to go toward a theory-based instruction for constructing a mental model in software design, the findings in this study can be used as a stepping stone for the other studies.

Mental models in general help people with understanding, predicting, and reasoning (Gentner, 2002). The findings of this study suggest that different mental models posit situations for different cognitive processes. That is mental models do not facilitate understanding, predicting, and reasoning evenly. Complex problem solving tasks will require the solvers to utilize different abilities at different stages. This explains why experienced problem solvers have multiple mental models (Markman & Gentner, 2001), but novices usually do not address either the abstract or local aspects of a problem.

Future Research

This study, because of its qualitatively exploratory nature (Gall, Gall, & Borg, 2007), does not support generalization about how all students learn mental models in software design. It does, however, provide insights in some areas from which I will make my recommendations for future studies.
This study demonstrates that novices have at least two different types of mental models: one is UI-based, and the other is programming-based. Novices also seem to have a shift in preference from UI-based to programming-based after learning Java language for one semester.

The relationship between the type of mental models and students’ software design performance remains unclear, although the experts did not seem to use the UI-based model. Assuming one mental model type is superior to the others, can instruction facilitate and accelerate the acquisition and use of the superior model? In addition, because experts create and employ several mental models during a complex problem-solving situation, it is not yet apparent whether or not a UI-based or a programming-based model might be suitable for certain parts of the entire problem-solving process. If that is the case, then how can it be determined which parts of the process each helps?

To be able to make a judgment on multiple solutions, designers must be able to anticipate the outcomes of each solution. This cognitive activity of predictive and comparative evaluation needs to be supported by an intensive body of knowledge. Experts have a knowledge repository that supports their evaluation processes, which novices by definition do not have. After one semester of training on programming skills, half of the novices (Benjamin, Caleb, Lazarus, and Mary) did more evaluation processes than they did previously. Improving programming skills might contribute to the increase of evaluation skills, especially Lazarus and Mary, who both changed from a UI-based mental model to a programming-based one.

So, should we simply accept that evaluation is a byproduct of using different mental models? We know experiential growth can facilitate the development of more robust mental models, which in turn improves reasoning and evaluating skills (Markman & Gentner, 2001), and we know that evaluation is important and is closely related to software quality. Maybe we can facilitate the acquisition of evaluation skills through teaching, and teach students early so that they can create programs of good quality. Not only the evaluation skills, but also other cognitive
processes should be examined. Teaching interventions like providing explicit instructions to facilitate decomposition skills and examples of different qualities with experts’ feedback can be further investigated to measure their effectiveness in forming mental models and using cognitive processes in design.

**Summary**

This work examined eight novices’ cognitive processes and mental models over one semester through three design sessions. Two showed a shift from UI-based mental models to programming-based models. The programming-based mental model allows them to be more focused on the underlying structures and procedures, which is more similar to what experts do. Other novices used some programming knowledge, but the effect on their mental models did not appear significant. Changing mental models also affects the type of cognitive processes the novices attend to. Although a previous study (Jeffries et al., 1981) discussed the processes involved in software design as primarily about decomposition, the novices in this study have shown a variety of different cognitive processes (problem comprehension, decomposition, structuring, mental simulation, and evaluation), both quantitatively and qualitatively. Even with one semester of C++ programming experience, the novices had very shallow domain knowledge of software design in the beginning. However, by the end of the study, deeper understanding and more robust representations started to emerge. Evidence of this can be seen in the novices using more detailed decomposition, longer time on task, and more evaluation skills.

Because these improvements are not observed globally, there are opportunities to improve design knowledge through teaching. Novices are capable of and ready for embracing those improvements along with learning programming knowledge. Different teaching strategies can be further investigated. For example, a problem-based learning environment might increase
novices’ performance in reasoning, and direction instructions such as verb/noun might help comprehension and decomposition. The development of cognitive processes and mental models for novices with about one year of programming experiences can be taught and should be addressed early.
REFERENCES


APPENDIX A:

RESEARCH CONSENT FORMS

Expert Participants

Informed Consent Form for Social Science Research
The Pennsylvania State University

Title of Project: Mental Model of Software Design for College Students of Different Programming Experience

Principal Investigator: Kuo-Chuan Yeh
204U Wagner Building
University Park, PA 16803
1-814-863-2888
martinyeh@psu.edu

Advisor: Dr. Kyle Peck
277 Chambers Building
University Park, PA 16803
1-814-865-2525
kpeck@psu.edu

1. Purpose of the Study:

The purpose of this research is to understand the programmers’ point of view of software design and software design behavior. People use different strategies for solving problems and it is partly because of the way they perceive the problem. Therefore, understanding how people view a problem is important to understanding how a problem is solved and why problem solvers approach a problem differently.

2. Procedures to be followed:

You will be asked to describe computer programming experience in an audio recorded interview. The interview will take only about 5 minutes. Then you will be asked to design a software system, using a paper and pencil to draw diagrams or write the description of the system. During the software design sessions, you will be asked to talk aloud about
what you are thinking, and describe the reason behind your actions. The design sessions will be video recorded with the camera pointing to the paper where your design will be. Your face will not appear in the video. The design session will not exceed 60 minutes.

After the design session when the video is transcribed by the primary investigator, you and the primary investigator will review the video together to clarify and further explain some of the design decisions. The video review session will not exceed 30 minutes.

3. Discomforts and Risks:

There are no risks in participating in this research beyond those experienced in everyday life. Some of the programming background questions are personal and might cause discomfort. You can choose not to answer the question if you do not want to.

4. Benefits:

The benefits to you include reflecting on your software design strategies. This may improve your learning process in programming and software design related courses.

The benefits to society include a deeper understanding about a programmer’s mental model of software design, which can improve the quality of computer science education in a software development related course.

5. Duration/Time:

A teaching interview for understanding the students' learning experience will take no more than 30 minutes. The design sessions will be no more than 60 minutes and the video review session which will not exceed 30 minutes. Therefore the total time given to this study will not exceed 2 hours.

6. Statement of Confidentiality:

Your participation in this research is confidential. The data will be stored and secured in the principal investigator’s home computer in a password-protected file. Penn State’s Office for Research Protections, the Social Science Institutional Review Board and the Office for Human Research Protections in the Department of Health and Human Services may review records related to this research study. In the event of a publication or presentation resulting from the research, no personally identifiable information will be shared.

Any data recorded in this study will be coded using pseudonym and your identity will be stripped from the data. The list of coding associated with your name and identity will be store separately in a password-protected file on the principal investigator’s password-protected personal computer. Your identity will not be identifiable from data without the list of coding. Only the principal investigator (Kuo-Chuan Yeh) and his advisor (Dr. Kyle Peck) will have access to the data.
The recordings will be destroyed after they are transcribed to text. The text will be stored in an electronic format and stored in the principal investigator’s password-protected personal computer for 5 years and then destroyed.

7. **Right to Ask Questions:**

Please contact Kuo-Chuan Yeh at (814) 863-2888 with questions, complaints or concerns about this research. You can also call this number if you feel this study has harmed you. Questions about your rights as a research participant may be directed to Penn State University’s Office for Research Protections at (814) 865-1775.

8. **Voluntary Participation:**

Your decision to be in this research is voluntary. You can stop at any time. You do not have to answer any questions you do not want to answer. You can also withdraw from this study without penalty. Refusing to participate or withdrawing early from the study will involve no penalty or loss of benefits you would be entitled to otherwise.

You must be 18 years of age or older to consent to take part in this research study. If you agree to take part in this research study and the information outlined above, please sign your name and indicate the date below.

You will be given a copy of this consent form for your records.

_______________________________________________  ____ _________________
Participant Signature      Date

_______________________________________________  ____ _________________
Person Obtaining Consent     Date
Novice Participants

Informed Consent Form for Social Science Research
The Pennsylvania State University

Title of Project: Mental Model of Software Design for College Students of Different Programming Experience

Principal Investigator: Kuo-Chuan Yeh
204U Wagner Building
University Park, PA 16803
1-814-863-2888
martinyeh@psu.edu

Advisor: Dr. Kyle Peck
277 Chambers Building
University Park, PA 16803
1-814-865-2525
kpeck@psu.edu

1. Purpose of the Study:

The purpose of this research is to understand a programmers’ point of view of software design and software design behavior. People use different strategies for solving problems and it is partly because the way they perceive the problem. Therefore, understanding how people view a problem is important to understanding how a problem is solved and why problem solvers approach a problem differently.

2. Procedures to be followed:

You will be asked to describe your experience about computer programming during an audio recorded interview. The interview will take only about 5 minutes. Then you will be asked to design a software system, using paper and pencil to draw diagrams or write the description of the system 3 times in three different sessions. During the software design sessions, you will be asked to talk aloud about what you are thinking and describe the reason behind your actions. The design sessions will be video recorded with the camera pointing to the paper where your design will be. Your face will not appear in the video. The design session will not exceed 60 minutes.

After the design session when the video is transcribed by the primary investigator, you and the primary investigator will review the video together to clarify and further explain
some of the design decisions. The video review sessions will not exceed 30 minutes per session.

The recordings, both audio and video, will be stored in a password-protected computer until it is fully transcribed. After they are transcribed, the recording will be deleted. Identities will be stripped from the transcriptions and the list of association of transcriptions to participants will be stored in a password-protected database, which is in a password-protected computer. Only the principal investigator (Kuo-Chuan Yeh) and his advisor (Kyle Peck) will have access to the data. The transcriptions will be stored for 5 years and destroyed in 2013.

3. Discomforts and Risks:

There are no risks in participating in this research beyond those experienced in everyday life. Some of the programming background questions are personal and might cause discomfort. You can choose not to answer the question if you do not want to.

4. Benefits:

The benefits to you include reflecting on your software design strategies. This may improve your learning process in programming and software design related courses.

The benefits to society include deeper understanding about a programmer’s mental model of software design, which can improve the quality of computer science education in a software development related course.

5. Duration/Time:

You will participate in 3 design sessions and 3 video review sessions, which will not exceed 4.5 hours in total.

6. Statement of Confidentiality:

Your participation in this research is confidential. The data will be stored and secured at principal investigator’s home computer in a password-protected file. Penn State’s Office for Research Protections, the Social Science Institutional Review Board and the Office for Human Research Protections in the Department of Health and Human Services, may review records related to this research study. In the event of a publication or presentation resulting from the research, no personally identifiable information will be shared.

Any data recorded in this study will be coded using pseudonym and your identity will be stripped from data. The list of coding associated with your name and identity will be stored separately in a password-protected file on a password-protected computer. Your identity will not be identifiable from data without the list of coding.

7. Right to Ask Questions:
Please contact Kuo-Chuan Yeh at (814) 863-2888 with questions, complaints or concerns about this research. You can also call this number if you feel this study has harmed you. Questions about your rights as a research participant may be directed to Penn State University’s Office for Research Protections at (814) 865-1775.

8. **Payment for participation:**

You will receive $100 at the end of the study when all 3 design sessions and 3 video review sessions are finished. The payment will be prorated in such a way that you will receive $30, $30, and $40 for the completion of each study session.

Total payments within one calendar year that exceed $600 will require the University to report these payments to the IRS annually. This may require you to claim the compensation that you receive for participation in this study as taxable income.

9. **Voluntary Participation:**

Your decision to participate in this research is voluntary. You can stop at any time. You do not have to answer any questions you do not want to answer. You can also withdraw from this study without penalty. Refusing to participate or withdrawing early from the study will involve no penalty or loss of benefits you would be entitled to otherwise.

You must be 18 years of age or older to consent to take part in this research study. If you agree to take part in this research study and the information outlined above, please sign your name and indicate the date below.

You will be given a copy of this consent form for your records.

_____________________________________________  _________________
Participant Signature      Date

_____________________________________________  _________________
Person Obtaining Consent     Date
APPENDIX B:

RESEARCH INSTRUMENTS

Demographic Survey

Software Design Study - The Pennsylvania State University

Name: ____________________________   Age: _____________________________

Major: _______________________________

For how many years have you been enrolled in college: _________________________

For how many years have you been using computer: __________________________

Please describe your typical computer use (e.g. word, internet, email … etc)

What programming languages do you have experience with? As part of the description, please indicate your skill level based your judgment.

What is the most complex software design project you have even done before? When was it? Describe how you did it.

Describe in detail how you would design a complex software system that you are not familiar with. (For example a UPS global package delivery system, or a flight pilot cockpit simulation system.)
Design Scenario

ONLINE LIBRARY MANAGEMENT SYSTEM

Background

A library has hired you to design an online library management system. This system will be used by librarians and patrons. Librarians use this system to check-in and check-out books, check book records, and send notifications when required. Patrons can use the system to look up books, view basic information, reserve/renew library items, and his/her own personal records.

Design Goal

Your goal is to design this system so that it can be used both locally and remotely. All components should work together to achieve all the required functionality. You should make the design as less redundant as possible.

Tasks for Patron

Look up library items and personal records, reserve and renew library items, change personal information.

Tasks for Librarian

Check-in and check-out items for patrons, add/edit/remove items, lookup overdue items, find patron based on checked out item, list items checked out by patron.

INSTRUCTIONS

(Remember, your performance WILL NOT be judged)
• Use the provided pencil, eraser, writing pads. You can use as many pages as you like.

• If you want to start a new design, DO NOT discard the old design, simply turn the page and continue.

• There is no time limit for how long you can design this system, normally 60 minutes. But try at least for 20 minutes if you do not know how to design such a system. Stop when you cannot make the design any better or you think it is good enough.

• You can use any symbols, draw lines, descriptions to represent your design.

• You should describe the system as much, detail as possible but stop when you think it should be handed over to programmers for implementation.

• You can combine components when you think it is appropriate. The design task description does not necessarily suggest how you should design this system.

• If you have questions about the library system, make modifications that you think are the most appropriate.
Post Interview

Sample Video Review Session Questions

Reflecting on the design session you did a couple of days ago, explain:

- (Show the design to the participant) Did you think about using other design options, the whole system or just part of it? If you did, describe other options you thought about and why you did not use them.
- If you thought about other options, what made you select your current design?
- Why did you combine those two functions?
- How did you know they could be combined?
- What advantage did you gain by combining these functionalities?
- Why did you separate the component into two different sub-components?
- How did you know it was better to separate that component?
- What advantage did you gain by separating these functionalities?
### APPENDIX C:
SAMPLE TRANSCRIPTIONS AND EPISODES

#### Expert Participant: E1

<table>
<thead>
<tr>
<th>Episodes</th>
<th>Themes</th>
</tr>
</thead>
<tbody>
<tr>
<td>A library has hired you to design an online library management system. The system will be used by librarians and patrons. Librarians will use the system to check in and check out books, check book records, and send a notification when required. Patrons can use this system to look up books, view basic information, reserve, renew librarian items, and his or her own personal record. Ok. Um, you goal is to design the system so that I can be used both locally and remotely. All components should work together to achieve all the required functionality. You should make the design as less redundant as possible. So, tasks, ok we’ve got task for patron, look up librarian items and personal records, reserve and renew librarian items, change personal information. Tasks for librarian, check in and check, add, edit, remove items, look up overdue items, find patron based upon checked out item, list items to be checked out by patron.</td>
<td>problem comprehension</td>
</tr>
<tr>
<td>Ok. So this system is straight forward um, um, no immediate questions about that. Um, instructions.</td>
<td>problem comprehension</td>
</tr>
<tr>
<td>Remember your performance will not be judged. Use a pencil, eraser, and paper to design the system, you can use as many pages as you, as you like. If you want to start a new design, do not discard old design, simply turn the page and continue. There is no time limit, normally 60 minutes, try at least 20 minutes if you do not know how to design such a system. Stop when you cannot make the design any better, or you think it’s good enough. You can use any symbols, draw lines, descriptions, to represent, um, you should describe this system as detail as possible, but stop when you think it should be handed over to programmers for implementation. You can combine component when you think it is appropriate. The design task description does not necessarily suggest how you should design the system. If you have questions about the library system, make modification that you think it’s the most appropriate. Ok, um, then that’s it.</td>
<td>non-design</td>
</tr>
<tr>
<td>Ok. Well the first thing I am gonna take my jacket off.</td>
<td>non-design</td>
</tr>
<tr>
<td>Ok, so, let’s start with the context diagram. Um, we have patrons, librarians, um, then we have this concept called library, library has um, has um, um.</td>
<td>decomposition</td>
</tr>
<tr>
<td>This is where it gets tricky cuz there is book, but multiple books under the same record, cuz we have multiple copies. So um, we’ll call this book um, just call it books for the context diagram.</td>
<td>evaluation</td>
</tr>
<tr>
<td>Um, a patron can be associated with the book, certainly with library, a librarian is associated with the library.</td>
<td>structuring</td>
</tr>
<tr>
<td>Then we got functions and the specific functions that we’ve got um, oh, patrons also have um, personal information, connected to the library, connected to the patrons.</td>
<td>structuring</td>
</tr>
<tr>
<td>Ok, so now, what can people do? Um, well, a librarian can check in slash out, that’s it’s to that. Um, a librarian can also, and I am gonna connect this to patron as well, add, edit, remove, um, so, um, now we get this problem that once a book is checked out.</td>
<td>decomposition</td>
</tr>
</tbody>
</table>
### Novice Participant: Caleb

<table>
<thead>
<tr>
<th>Episodes</th>
<th>Theme</th>
</tr>
</thead>
<tbody>
<tr>
<td>Okay. I decided to start off with the librarian part of it because I think that they need to be able to use the information before the patron needs to use the information. So I'm gonna start there.</td>
<td>problem comprehension</td>
</tr>
<tr>
<td>And, um, the first thing they'll need to know would be the if the book is in or not.</td>
<td>problem comprehension</td>
</tr>
<tr>
<td>So there would need to be a system to check if the book is in.</td>
<td>decomposition</td>
</tr>
<tr>
<td>So the availability of the book {writing this on the paper}. And then it would have to be whether it’s in, out.</td>
<td>decomposition</td>
</tr>
<tr>
<td>And I would also have to know who would have it or when it’s coming back in.</td>
<td>decomposition</td>
</tr>
<tr>
<td>So they need to be able to check on the overdue books, which would then also connect with the who or when, which I will make a whole separate thing or who has the book.</td>
<td>decomposition</td>
</tr>
<tr>
<td>Because that’s going to hold a lot more information than just who or when that will have, that will be contained with when, how long, etc</td>
<td>structuring</td>
</tr>
<tr>
<td>So the overdue would need to access those book and availability itself.</td>
<td>structuring</td>
</tr>
<tr>
<td>It would also just access whose book as that has the primary information as to whether it’s available or not.</td>
<td>structuring</td>
</tr>
<tr>
<td>The whose book will also contain . . . has to be able to have access to the, the book itself for them to be able to look up and the book or who has the book by the title. So it can go back and forth that way.</td>
<td>decomposition</td>
</tr>
<tr>
<td>I guess it has, it has to have who as well for that same reason so it can go back and forth between the two.</td>
<td>decomposition</td>
</tr>
<tr>
<td>The librarian will also have to have a separate thing just for checking in and checking out books.</td>
<td>decomposition</td>
</tr>
<tr>
<td>And, and that will take that information and store it into the who’s book as well. You know those are separate . . . those will need to be readily accessible because they are the primary things that the librarian is doing.</td>
<td>structuring</td>
</tr>
<tr>
<td>The availability would also have to be tied into the checking in because . . . no checking out . . . sorry . . . because to be able to check the book out it has to be available for the patron to take the book out…..</td>
<td>structuring</td>
</tr>
<tr>
<td>I’ll do the patron for now and tie them both together.</td>
<td>decomposition</td>
</tr>
<tr>
<td>The patron needs a system to look, to look up the books. And they would need to be able to look it up by the, the title of the book, author, the call number just for ease of use for the patron himself.</td>
<td>decomposition</td>
</tr>
<tr>
<td>And then in the look up they will need to be able to either reserve the book or check out the book and even . . . they can even check if it’s available. Just if they’re not ready to check it out or reserve it now then they can see if the book’s available, which can be the same there as it, the availability as the librarian thing.</td>
<td>decomposition</td>
</tr>
<tr>
<td>They’re also going to need their account information so that they can have like their personal info just even name, their books checked out, any overdue fees.</td>
<td>decomposition</td>
</tr>
</tbody>
</table>
VITAE

Education

Ph.D. in Instructional Systems, Pennsylvania State University, University Park, PA. 2009.
Graduate course work, Management Science and Information systems, Pennsylvania State University, 2002 to 2003.
B.S. in Computer Science and Information Engineering, Tamkang University, Taipei, Taiwan. 1993.

Teaching

At Penn State University

Publications and Presentations

Yeh, K. (2005). What does one-to-one computer environment mean to teachers in an elementary school? Teacher’s perception of one-to-one laptop computers in classrooms. Association for Educational Communications and Technology (AECT), Orlando, FL.
Peck, K., Popp, J. D., Haughton, N., & Yeh, K. (2001). PT3 Database Project. Association for Educational Communications and Technology (AECT), Atlanta, GA.

Experience

Web Support Graduate Assistant, Information Technology Services, Penn State University, 2006 to 2009.
ActionScript Developer, World Campus, Penn State University, 2006 summer (12 weeks).
Learning Skill Coordinator, University Learning Center, Penn State University, 2005 to 2006.
Graduate Assistant, TICKET certificate program, Penn State University, 2004 to 2005.
Web Application Developer/System Administrator, PT3 Project, Penn State University, 2000 to 2004.
Software/Firmware Developer, Skanhex Technology, Taiwan, 1997 to 1999.
Software/Firmware Developer, Avigramm Technology, Taiwan, 1997.
System Engineer, Total Integra Technology, Taiwan, 1995 to 1996.

Organizations

Member of Association for Computing Machinery
  • ACM Special Interest Group in Computer-Human Interaction. (ACM SIGCHI)
  • ACM Special Interest Group in Computer Science Education. (ACM SIGCSE)
Member of Society for Information Technology and Teacher Education (SITE)