COGNITIVE ASPECTS OF SOFTWARE ENGINEERING PROCESSES

by

SHAOCHUN XU

DISSERTATION

Submitted to the Graduate School

of Wayne State University,

Detroit, Michigan

In partial fulfillment of the requirements

for the degree of

DOCTOR OF PHILOSOPHY

2006

MAJOR: COMPUTER SCIENCE

Approved by:

_____

Advisor Date

_____

_____

_____

_____

DEDICATION

To my parents and my family for their support.

ACKNOWLEDGMENTS

This thesis is the result of four and half years of work which I have been accompanied and supported by many people. I express my gratitude to all of them.

The first person I would like to thank is my supervisor, Prof. Dr. Vaclav Rajlich. I joined the SEVERE Group (**S**oftwar**E** **V**isualization and **E**volution **RE**search Group) at the beginning of 2002. During the years since, I have seen the sympathetic and principled manner which Vaclav has helped me and others here at Wayne State. His enthusiasm and his viewpoint on research and his idea for providing only high-quality work have made a deep impression on me. I give my gratitude for his having shown me this methodology of research. I trust he understands how much I have learned from him. I also would like to mention his encouragement and support in selecting this thesis topic. Since research on cognitive activity in software engineering processes has been halted for a few years due to difficulty to find a better approach, he warned me of the dangers of dealing with such a topic in Ph.D. thesis. Nevertheless, he continually discussed the issues and the directions with me, and I greatly benefited from his guidance and input. I thank him for his patience and guidance during my research activities.

I would like to thank Dr. Andrian Marcus for his help and his comments on my thesis. I would also like to thank these members of my Ph.D. committee who monitored my work and took time and effort in reading the thesis and for providing me with valuable comments about it: Dr. Weisong Shi, Dr. Daniel Grosu, and Dr. Dale Brandenburg.

I also thank all other members of SEVERE group for providing useful discussions on my emerging ideas: Joe Buchta, Yong Jiang, Andrey Sergeyev, Denys Poshyvanyk, Travis Xie, Dapeng Liu, Maxym Petrenko. I am also grateful for Prof. Rajlich and the Department of Computer Science for its financial support while I was a Teaching Assistant during my study in Wayne State University.

I am grateful for my colleagues in Sault Ste. Marie, Ontario, Canada, Dr. Jay Rajnovich, Prof. Gerry Davies, and Dr. George Townsend for their help in writing this thesis.

I had the pleasure to supervise and work with a group of graduate students and undergraduate students who were joining the experiments. Without their help, this work could not have been done.

Finally thanks to Ms. Shu Yan, Mr. James Troescher, who spent their precious time to review my thesis and provided useful suggestions in correcting this thesis.

TABLE OF CONTENTS

LIST OF TABLES

TABLE **PAGE**

LIST OF FIGURES

FIGURE **PAGE**

CHAPTER 1

INTRODUCTION

1.1 Background

Software is a human-intensive technology and the studies of cognitive processes in software engineering can shed light on many software engineering problems [125] [39]. Software engineering deals with distinct processes, such as requirements analysis, software design, software evolution, program debugging, software reuse, software maintenance and program documentation. A common characteristic of all these processes is in the method employed in handling a large amount of knowledge that is distributed over many knowledge domains, such as the problem domain, program domain, and programming techniques domain [31]. The program itself is also a large repository of this knowledge and may contain knowledge that is not available elsewhere, as documented in [86]. It also contains knowledge of all design decisions that were made during the program development and the consequent program evolution [129].

When either evolving or maintaining the program, it is necessary to recover that knowledge; otherwise maintenance or evolution will be impossible. It is also necessary to communicate the knowledge to all new programmers who are engaged in software project. Although the knowledge is embedded in the program, it cannot be easily recovered since the bits of knowledge are delocalized into different parts of the program. Moreover, the consequences of the decisions, rather

than the decisions themselves, appear in the code. In many ways, the recovery of knowledge from the code is similar to that of solving a puzzle, in that it is laborious and error prone. Some knowledge might not be able to be recovered. Loss of programming knowledge was identified as a leading cause of code decay [28].

Forward engineering is used to process the domain knowledge and turn it into design decisions that will be reflected in the source code (i.e. encode the knowledge into the code), whereas program understanding and reverse engineering are used to decode this knowledge from legacy systems and their documents. An example of forward engineering is the process of software development. Therefore, the cognitive process is the center of all software engineering processes to such extent that understanding the cognitive process and documenting the knowledge during software engineering processes becomes a critical and practical issue.

Empirical studies have been applied in software engineering research [9]. Some of these issues are of great importance, including how to conduct the experiments, how to capture the data, and how to analyze the data [83]. Without an effective way to collect and capture the data, the data collected might be incomplete or incorrect. Conversely, an accurate coding scheme is critical to analyze the data and to explain the experiment result.

The empirical approaches, which have been commonly used by cognitive scientists, include protocol analysis, task analysis, and discourse analysis [51, 58, 133].

Think-aloud protocol analysis, one of protocol analysis techniques, uses a single subject who verbalizes his/her thoughts while performing a task. Think-aloud

protocol analysis has been used in social sciences for many years and is now increasingly common in software engineering research [158]. However, there are some deficiencies in this approach. These include the placebo effect [127] and the Hawthorne effect [1]. For that reason, the researchers have to search for additional empirical methods that may offer relief from these two problems.

There are many cognitive theories which have been applied and used in studying cognitive process in software engineering, including text understanding [4], knowledge modeling, problem-solving and learning. Particularly in the area of program comprehension, many theories have been proposed. Examples of these include the top-down, and bottom-up theories [24]. However, previous theories and models might not be able to explain the important aspects of software engineering process, and quite a lot cannot be used to explain well the differences between expert and novice programmers. Therefore, a new model should be developed to meet such a need.

There is some research on how experts perform programming tasks (such as program comprehension, program debugging etc) that differ from novices. However, none of this research was performed from the point view of actual cognitive activities.

1.2 Motivation and Hypotheses

In order to better understand the cognitive aspects of software engineering and the nature of program knowledge, several approaches should be taken: 1) we

study in detail the most common protocol research method: think-aloud protocol, to see the advantages and disadvantages, and to see if we can improve that method in order to collect more complete and accurate empirical data; 2) we study the existing recording methods such as tape recording and videotaping, to find a better way to document the programming process; 3) we study the existing cognitive theories in order to build a more reasonable and useful model to explain the cognitive process in software engineering 4) then we use our new approach (including new protocol, new recording method, and new code scheme) to conduct a pilot study and further case studies on selected software engineering processes (incremental software development, program debugging, software evolution), in order to understand the cognitive process and to evaluate our new approach.

The type hypotheses for this research are:

1. "There are better ways than think-aloud protocol to study the cognitive activities in software engineering"
2. "The combination of several learning theories can be used to better study the cognitive activities used during software engineering process"
3. "Intermediate programmers follow a similar cognitive process during incremental software development"
4. "Experts and intermediate programmers follow different cognitive processes when they perform incremental software development"
5. "The method we use for incremental software development can be also applied in studying software evolution and program debugging"

1.3 Contributions

We studied the think-loud protocol and discovered several defects. Based on the idea of pair programming [10], we proposed a dialog-based protocol for cognitive research in software engineering, since the majority of software engineering tasks can be done with pairs. In pair programming, where two programmers work on a programming problem using only one computer, the technique requires a constant dialog between them. This dialog can be unobtrusively recorded to provide evidence for programmer cognitive process. The dialog-based protocol can greatly reduce the negative effects of the think-aloud protocol.

We also discovered a new recording method in which screen-capturing software with voice recording can provide more complete and higher quality data. Software screen-capturing records the programming process and communication between the two programmers.

We extended the constructivist learning theory and refined the cognitive process into four cognitive activities: absorption, reorganization, denial and expulsion. By combining the extended constructivist learning theory and Bloom's taxonomy of cognitive domain, we proposed a new cognitive model called self-directed learning. The extended constructivist learning classifies the cognitive activities and the Bloom taxonomy assesses their complexity with six levels (knowledge, comprehension, application, analysis, synthesis, evaluation). The self-directed learning theory provides an encoding scheme used to analyze the

empirical data. This model forms a basis for study of the cognitive process involved in various software engineering processes.

With the unified cognitive model, we can study the distribution of the four cognitive activities and six Bloom levels in order to distinguish various software engineering processes. We can also use this model to understand the cognitive part of these processes and to improve software engineering practice through this better understanding. It can also be used to study the cognitive differences and gaps between novice and expert programmers during their various software engineering activities.

We used our new empirical approach, including the dialog-based protocol, software screen-capturing, and the self-directed learning theory, to study cognitive activities by conducting experiments on 10 intermediate level pairs. We gave particular attention to the differences between theirs' and the experts' learning. Compared to intermediate programmers before starting to write their code, experts discussed more domain concepts and were more willing to reconsider and correct obsolete design decisions. Experts mostly concentrated on one concept at one time, while intermediate programmers often discussed several concepts simultaneously. We also found that the cognitive process applied by experts is different from that of intermediate programmers.

We used this approach to study program debugging. We confirmed that the self-directed learning model can also be used to study the cognitive process during program debugging.

We studied pair programming used in software evolution in graduate software engineering project. We applied the pair programming in software evolution project by asking students to make incremental changes in an open project. We found that paired programmers completed the change request task with higher quality and shorter time compared to programmers who worked individually.

We think that our empirical study can also provide some knowledge for tool development, i.e. it could lead towards a new generation of documentation systems that could be better used to capture the program knowledge during software development and maintenance, and in turn improve program understanding and reduce maintenance costs. This approach can also provide some guidance for education and training purposes.

This study will also shed some light on the different philosophical understandings (learning view) of software engineering processes, which in turn improve the efficiency of software development, program debugging and program comprehension, and the quality of software produced.

1.4 Organization of the Dissertation

This dissertation consists of nine chapters. Figure 1.1 represents the roadmap to the dissertation. The rectangles represent the chapters of the thesis; the arrows represent the relation between the chapters.

Chapter 2 reviews the related work. It provides a summary of previous work in cognitive research in incremental software development, program debugging and

program comprehension. We also reviewed cognitive research methods, such as think-aloud protocol, recording methods and code schemes. The review on related cognitive theories, particularly the constructivist learning and Bloom's taxonomy of cognitive domain is also included in this chapter.

Chapter 3 describes the methods used for our research. A novel empirical research method: dialog-based protocol is given in detail. We also make a detailed comparison with think-aloud protocol. In chapter 3, we also explain the recording methods which are used for our experiment, including videotaping and software screen–capturing.

Chapter 4 outlines the self-directed learning model, a new code scheme, based on extended constructivist learning theory and Bloom's taxonomy. A comparison with some existing models is also discussed.

Chapter 5 covers the detailed case study on incremental software development, where we apply our novel empirical approach. We describe the detailed case study design and case study result.

Chapter 6 describes the case study on program debugging, in which we study the cognitive process using the self-directed learning theory.

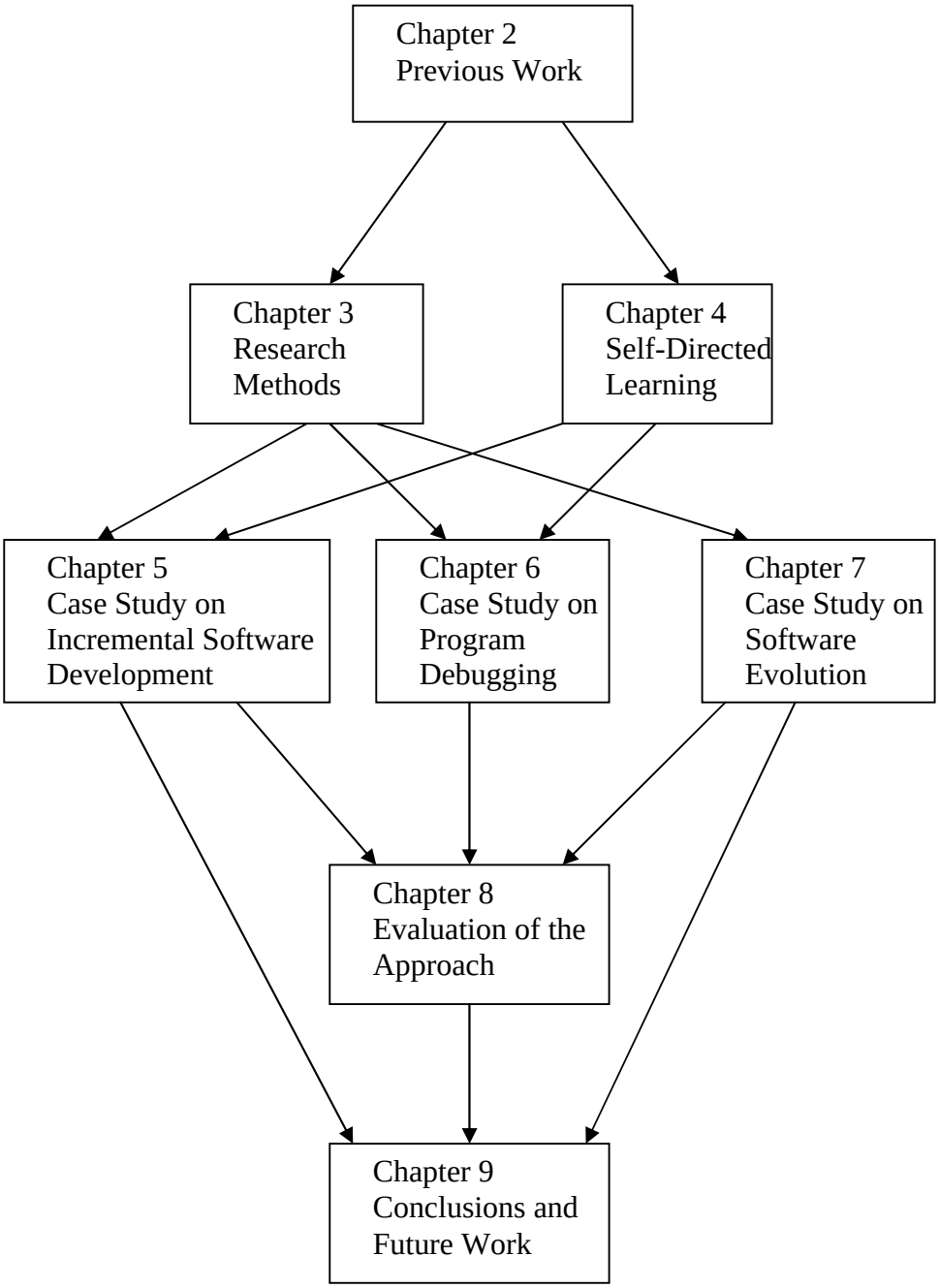

Figure 1.1: Roadmap to the thesis

Chapter 7 covers the case study on software evolution, in which we conduct experiments with pair programming and solo programming, and then compare the experimental results. We also discuss the survey result on pair programming in this chapter.

Chapter 8 surveys the strengths and weaknesses of our empirical method, including the dialog-based protocol, software screen-capturing and self-directed learning theory.

Chapter 9 concludes the thesis. It summarizes our research results and provides some potential future work.

Three appendixes are attached in the thesis. One is for the pilot study on incremental software development; one for all the detailed data for the case study on incremental software development; one for the case study on program debugging.

CHAPTER 2

PREVIOUS WORK

2.1 Cognitive Research in Software Engineering

The process of cognitive study of programming began during the 1970s. At that time, computer scientists were interested in testing or evaluating new programming tools by using methods from experimental psychology. One example is when Shneiderman (1976, 1980) [134, 136] proposed two independent variables to measure the performance when modifying a program. One was the use of meaningful mnemonic variable names and the other was the use of comments. His experimental result showed that the more complex the program, the more the use of mnemonic variable names helped in understanding it and the more functionally descriptive comments use, the faster the conversion of code to internal semantic structure [134, 136] [135].

The objective of the cognitive research approach during that period was to enrich the theoretical frameworks by borrowing theories from cognitive psychology and other disciplines [23]. Many theoretical frameworks were proposed during the time period between 1970s and 1980s. These can be classified into four categories.

- Understanding of natural language text: A few theories were proposed to explain the understanding of programs. For example to understand programs, Atwood and Ramsey applied the theory of Kintsch and van Dijk (1978) [4] and found that understanding a program is similar to understanding a text that includes

recognizing letters and words, syntactic parsing of sentences, understanding the meaning of words and sentences, and incorporating the meaning of the text in other present knowledge.

- Learning: Cognitive and educational models were borrowed to explain the learning of programs. Mayer (1981) [95] used the learning theory to explain how a novice learns computer programming. He stated that two things happen as novices learn to program. First, beginning programmers acquire new information about the programming language and the rules that govern it. Secondly, they create a mental model of what happens inside the computer as the program statements get executed.

- Modeling of knowledge: The schema theory was used to study how the experts organize their knowledge during program comprehension [141]. McKeithen et al. (1981) [97] applied the theory of information processing on knowledge organization in complex domains such as chess to the programming domain.

- Problem solving: Brooks (1977) [22] used the Newell and Simon (1972)'s theory of information processing [103] to study the cognitive mechanism in program design. He described the programming process as one of constructing mappings from a problem domain to an implementation domain, possibly through multiple levels. The work of Newell and Simon has long dominated the theories on problem solving in cognitive science.

During the first workshop on Empirical Studies of Programmers in 1986, one paper presented by Curtis [40] and one paper presented by Soloway [140] played a great role to lead to continued research on theoretic direction. Since that time,

several cognitive models on program comprehension and on other processes have been proposed. Included are the top-down model, bottom-up model and as-needed model [90] [24]. Detailed research history will be described below. The theories from other disciplines have recently also been applied in explanation of software engineering processes. For example, Rajlich (2002) [115] and Exton (2002) [52] applied constructivist-learning theory in program comprehension. Wang (2002) [164] created the term, Cognitive informatics (CI), which is a multidisciplinary research area at the intersection of cognitive science, neural psychology, philosophy, software engineering, and other disciplines. Wang (2004) [165] also investigated the relationship between software engineering and cognitive informatics and developed a set of laws and cognitive properties for software engineering.

In addition to the cognitive models proposed during this period, there was much work completed in studying the activity of novices and experts in conjunction with the participation of some professionals in experimental studies or field studies[154] [47]. These recognized the various differences between novices and experts during software development that included knowledge, strategies they use etc. The differences between novices and experts during debugging and program comprehension were also investigated [72, 153].

Since our work mainly concentrates on selected software engineering processes (incremental software development, program debugging, and software evolution), we summarize the research on cognitive activity and tools according to individual software engineering processes as follows.

2.1.1 Incremental Software Development

The classic software development process suggests a systematic sequential approach to software development that begins at the requirements specification level and progresses through analysis, design, coding, testing and maintenance. Although the incremental development process is been widely used, its history can be traced back to 1950s, when it was derived from several projects [87]. The incremental software development process has no sharp distinction between the requirements specification and design in the so-called elaboration phase, and with iterations during the construction phase. Incremental software development and subsequent program evolution is a process where the programmers add one program property at a time. Programmers often start with an existing program or a program framework covering a fundamental functionality, and add and delete properties to or from it.

Although Curtis (1988) [41] indicated that development of large software systems is composed of the activities of learning, communication and negotiation, not enough research has been done on the knowledge and cognitive process during incremental software development. A systematical analysis on the programming strategy was conducted by Davies (1993) [45], who suggested that what is needed is an explanation of programming skill that integrates ideas about knowledge representation with a strategic model, enabling one to make predictions about how changes in knowledge representation might give rise to particular strategies and to the strategy changes associated with developing expertise. Fisher et al. (1994) [53]

proposed a support for the incremental development based on a specific knowledge model, where seeding, evolution and reseeding are the three stages of knowledge capturing and transformation. Henninger (1997) [74] recognized that software development is a process involving various knowledge resources that continually change during the process. Ribbins (1999) [74] used several cognition theories, including Reflection-In-Action, to understand the design process and Wallenstein (2002) [162] proposed one framework for tool design based on distributed cognition theory. Ostward (1996) [109] discussed the social and evolutionary process involved in the knowledge construction during the software development process. Baragry (2000) [7] viewed the software engineering process using other disciplines such as metaphysics, epistemology, and psychology of cognition. In fact, software development is actually a process of knowledge construction. However, none of the above approaches evolve from constructivism and learning approaches. Recently, the analogy between constructivist learning and incremental software development process has been recognized by Rajlich and Xu (2003) [119], who identified four cognitive activities (absorption, denial, reorganization and expulsion) which correspond to incremental change, rejection of change request, refactoring and retraction, four programming activities.

The large volume of knowledge that must be manipulated during software development requires tool support. Although there are some suggested tools for program understanding and software design, most documentation tools generate documents after the programs have been completed, rather than documenting the knowledge on the fly during incremental development process. Some

documentation tools like Javadoc [81] extract knowledge from comments rather than from actual code. The design community has attempted to create methods and notations for capturing design information, but their attempts have failed to become standard practice [129]. One common approach is use of the expert system that requires domain experts to articulate their knowledge to create a knowledge base in general. This approach cannot capture specific knowledge used for the system, and therefore, will not be of much help in maintenance and evolution. Another approach is to design tools to document design decisions when they occur during design meetings. There are problems with this approach, the most obvious one being the excessive amount of time spent by designers in writing detailed notes about their discussion, and the problem of the notes being insufficiently rich to capture the full complexity of the design decisions.

Ribbins (1999) [122] used cognition theories as guidance in the development of design tool features. A tool was proposed to provide developers with efficient access to the group memory in a software development project [39]. Neither tool was built based on the cognitive activity, and therefore neither can capture the necessary and complete domain and programming knowledge. Knowledge can be more easily and correctly captured during the software development process. Such a tool should support the mental process of the incremental development and should be based on cognitive activities. Once we capture this knowledge, it will further accelerate software development. With the help of this captured knowledge, debugging and program maintenance would be made much easier.

2.1.2 Program Debugging

Debugging refers to the combined process of testing and code correction. Errors can be made in various stages of software development or evolution, such as specification, implementation, or debugging of earlier errors [84]. Programmers spend a large part of their efforts in debugging [21]. In spite of the progress in the past years, debugging still remains labor-intensive and difficult [3].

Program debugging involves a complex cognitive process and requires comprehension of both program behavior and program structure. It is one of the most challenging areas of software engineering. In turn, debugging requires program comprehension and testing as the central tasks [24] [137].

Vessey (1983) [153] and Gould (1975) [69] conducted research in the cognitive aspects of debugging and characterized debugging as an iterative process of synthesizing, testing, and refining hypotheses about fault locations and repairs. Araki (1991) [3] provided a general discussion on the framework of debugging and emphasized the process of developing hypotheses. Bisant and Groninger (1993)[17] considered that fault detection involved two processes: a comprehension process and a fault location process. Model-based debugging techniques have been proposed by Ducasee (1993) [49] who emphasized that different parts of the program are perceived by the user as having different levels of reliability.

Yoon and Garcia (1998) [179] investigated how tools can help programmers obtain clues about the bugs. Two debugging techniques were identified: comprehension strategy and isolation strategy. In comprehension strategy, the

programmer searches for the difference between the program requirements and the programs' actual behavior. In isolation strategy, the programmer makes hypotheses about the bugs and then searches for bug locations in the program. Van et al. (1999) [152] performed a field study on the understanding of software during corrective maintenance of large-scale software. They found that programmers work at all levels of abstraction (code, algorithm, application domain) at an approximately equal level and frequently switch between levels of abstraction. Three steps of the debugging process have been recognized by Uchina et al. (2000) [151]: program reading, bug location and bug correction. Mateis et al. (2000) [94] described a technique to diagnose errors involving objects and reference aliasing. Ko and Myers (2003) [84] have presented a model of programming errors. They found that most errors were due to attention and strategic problems in implementing algorithms, language constructs, and uses of libraries.

The common debugging tools are integrated into compilers, i. e., the integrated development environments (IDEs). The IDEs provide some ways to capture some of the language-specific predetermined errors (e.g., missing end-of-statement characters, undefined variables, and so on) without requiring compilation. Another way to help debugging is the visualization of the necessary underlying programming constructs as a means to analyze a program [5]. There are some researches which try to automate the debugging process through program slicing[78].

2.1.3 Program Comprehension

In software evolution or software maintenance, programmers modify or add specific program concepts or features based on the change request. In order to modify the program, the program understanding process should be performed first. Therefore, program understanding plays an important role during software evolution[28].

A large body of available research deals with program comprehension [24] [89, 90]. Various models have been established. According to the top-down theory of comprehension [24], the programmers first accept a hypothesis that describes the program and then attempt to verify this hypothesis. Further hypotheses may be required in order to build a hierarchy of hypotheses. Rajlich et al. (1994) [117] proposed a layer program comprehension strategy in which the programmer creates a chain of hypotheses and subsidiary hypotheses concerning the properties of the code and then looks for evidence (beacons) in the code. This approach is very close to the top-down model. After conducting an experiment on program comprehension, Fix et al. (1993) [55] presented five abstract characteristics of the mental representation of computer programs: hierarchical structure, explicit mapping of code to goals, foundation on recognition of recurring patterns, connection of knowledge, and grounding in the program text. His approach can also be classified into the top-down approach.

Letovsky (1986) [90] introduced the bottom-up theory, in which programmers gather small chunks of source code to form higher level abstractions. These abstractions are recursively grouped to produce a high level comprehension of a

program. Schneiderman (1979) [137] also considered that understanding a program involves the creation of multilevel internal semantic structures. This multilevel structure is created in a bottom-up manner, where until the entire program is understood, programmers understand the function of a group of statements and pieces them together to obtain higher levels of chunks. Further, Pennington (1987)[111] suggested that program readers build at least two mental models of the program they are studying, a program model and a domain model. The program model is characterized by an abstract knowledge of the program's text structures and the domain model relates objects and functions in the problem domain to source-language entities.

An as-needed approach was suggested by Littman et al. (1986) [91], where programmers look only at the code related to a particular problem or task. Similarly, after experiments on expert programmers modifying PASCAL programs, Koenemann and Robertson (1991) [85] thought that program comprehension is a goal-oriented and hypothesis-driven problem-solving process, In this concept, programmers used as-needed strategy rather than systematic strategy during program comprehension. Clement (1996) [33] proposed an approach, similar to as-needed approach, in which he suggested a narrowly-focused, reusable domain model to integrate and to aid in the process of top-down system comprehension.

Mayrhauser and Vans (1994) [157] found that maintenance programmers use a multi-level approach that switches between the three postulated areas (domain model, situation model, and program model) and the maintenance activities are described by a small set of cognitive processes which aggregate into

higher level process (on several level of abstraction). Von Mayrhauser and Vans (1998) [160] further pointed out that program comprehension is not a simple top-down or bottom-up process but rather that in which programmers apply various models depending on their tasks and their domain or programming knowledge. Programmers with more knowledge of domains more often prefer the top-down approach than those with less domain knowledge. Programmers with less programming knowledge are more likely to use the bottom-up approach in program comprehension. Based on their observations, Von Mayrhauser and Vans (1998)[160] proposed an integrated program comprehension model that combines top-down, bottom-up, and as-needed; any of the three sub-models may be used at any time during program comprehension process.

Soloway et al. (1988) [142] observed that the process of program understanding is a cycle that includes reading the code (obtain the knowledge), raising questions (apply knowledge), making a conjecture (synthesize and generate hypothesis) and searching through code for answers (evaluation). The process is also called the read-question-conjecture-search cycle.

A small knowledge base was proposed in order to help the apprentice navigate the software and help understanding the program [130]. Davis (1990) [43] studied the roles of programming plans and selection rules during program comprehension. Benedusi et al. (1993) [12] studied the role of test cases and human activities in program comprehension.

Semantic and syntactic knowledge comprehension was also studied by Shneiderman and Mayer (1979) [137]. They found that programmers retain both

semantic and syntactic knowledge in long-term memory, and that they use short-term and working memories in performance of various program-related tasks. Syntactic knowledge is programming language dependent, but semantic knowledge refers to algorithms or data structure. The chunk concept was first proposed by Davis (1984) [46] to represent the basic knowledge unit in the program. The knowledge unit in a program was also examined and classified into three categories by Clayton et al. (1998) [32]: knowledge type, design decision used, and the type of analysis required in uncovering the atom. Robillard (1999) [124] identified two types of knowledge in software programs: topical and episodic. Topical knowledge refers to the meaning of words and episodic knowledge consists of people's experience with knowledge.

Clayton et al. (1997) [30] described the application of a domain-based program understanding process, called Synchronized Refinement, to the problem of reverse engineering of the Mosaic World Wide Web browser software. The domain knowledge in program comprehension was also studied by Shaft and Vessey (1995) [132] and Rugaber (2000) [128]. Rugaber (2000) [128] even proposed a domain-based understanding approach and suggested a tool needed to store those domain knowledge for future maintenance and reuse.

Concept location is a key issue in program comprehension. Concept location constructs the mapping between programming concepts and the related software components [16]. Wilde et al. (1995) [168] developed a concept location technique called Software Reconnaissance that is based on analysis of program execution traces. Chen and Rajlich (2000) [28] proposed a method based on a search of static

code that is represented by Abstract System Dependence Graph and which consists of control flow and data flow dependencies among the software components.

Baniassad and Murphy (1998) [6] introduced the conceptual module approach that allows a software engineer to simultaneously perform queries about both the existing and the desired source structure.

Many tools have been designed and are available for program comprehension during this stage [138]. They can be classified into five categories:

- Language-based tools: emphasize a specific language and its problems.
- Paradigm-based tools: focus on a single programming paradigm.
- Dependency-analysis based tools: maintain static/dynamic program dependencies.
- Hypertext-based tools: use hypertext techniques to understand structural/behavioral aspects of programs.
- Program-animation based tools: use animation to understand the behavior of a program.

It is worth to notice that the cognitive approach has been used to guide the tool development for program understanding [180]. Singer et al. (2005) [139] demonstrated a tool to support browsing through software that kept track of the navigation history of software developers. Storey et al. (1999) [147] examined various cognitive models and suggested some issues to guide the development of tools that could be used to help in software exploration. However, their tools are for

software evolution only; hence, they do not capture the complete domain and programming knowledge.

2.1.4 Novices and Experts

Soloway and Ehrlich (1984) [141] observed that during software development, experts employ high-level plans while novices use more lower-lever ones. Koenermann and Robertson (1991) [85] emphasized that experts primarily use the top-down strategy and novices often use the bottom-up strategy.

A field study was performed by Visser (1987) [154] on professional programmers to study the strategies used when they were doing programming. He found that programmers used a number of data sources and included program listings into them, so programmers may recall that a solution exists in a listing, find the listing, and then use the coded solution as a plan for the current problem.

Several other research projects deal with the differences between novices and experts during debugging procedures. Gugerty and Olson (1986) [72] and Vessey (1983) [153] found that the knowledge differences between novices and experts on the program and programming explain why experts can debug more accurately. Spohrer and Soloway (1986) [145] and Gugerty and Olson (1986) [72] observed that novices often introduced more new bugs to the programs than do experts.

The differences between novices and experts during program comprehension have been well studied [13, 111]. Holt et al. (1987) [77] examined

programmers' cognitive representations of software by making either simple or complex modifications to the program using three different design methodologies. Berlin (1993) [13] provides some insights into the differences between experts and novices during program comprehension. The novices were probably well-trained professionals but without the specific domain and/or system specific knowledge the experts had. Berlin (1993) [13] also discussed the types of knowledge, and differences that distinguish the experts from the novices and also discussed how this may factor into productivity differences. His observations indicate the multi-faceted knowledge needed for real-life programming expertise, and the knowledge and skills that make experts so much more effective in their daily work. He also showed that one major reason that makes good programmers more productive is that they know who to ask about particular problems and are able to quickly refer to other peer colleagues.

Burkhardt et al. (1998) [26] analyzed object-oriented (OO) program comprehension by experts and novices in three dimensions of comprehension strategies: the scope of the comprehension, the top-down versus bottom-up direction of the processes, and the guidance of the comprehension activity. They found out that experts used top-down, inference-driven strategies, while novices rarely used the top-down approach, but instead, used the execution-based guidance procedure. Wiedenbeck (1999) [167] studied the comprehension process of novice programmers on small procedural and object-oriented programs. She found that novices tend to develop a more function-related mental representation with less emphasis on data flow.

Detienne (1990) [47] conducted experiments on professional programmers in order to study the difficulty with OO language. As one of the earliest empirical studies of impact of OO programming, it was perhaps instrumental in initiating further work in this area. However, Detienne's study has some limitations and therefore, there are doubts concerning the validity of its argument regarding the inadequacies of the OO approach.

Fix et al. (1993) [55] studied how novices and experts understand Pascal programs and found that expert programmers use more abstract mental representations for the programs than do novices. Studies of differences between procedural and OO programmers were carried out by Corritore and Wiedenbeck (2000) [36] and Ramalingam and Wiedenbeck (1997) [121]. They found that the OO programmers tended to use a strong top-down approach to program understanding during an early phase of study of the program, but increasingly used a bottom-up approach during the maintenance tasks. Further, the procedural programmers used a more bottom-up orientation throughout all activities [37] [38] [36]. Mosemann and Wiedenbeck (2001) [100] found that novice programmers used a sequential or control flow view of the program during program comprehension and had difficulties with a data flow view.

Perkins and Martin (1986) [112] described the main difficulties faced by novices in general as "fragile knowledge" and "neglected strategies". Novices obtain the knowledge, but do not know how to apply it. Gilmore (1990) [66] also studied the programming knowledge of experts and realized that expert

programmers possess many programming strategies, while novices suffer from lack of both knowledge and adequate strategy to cope with programming problems.

Other differences between novice and expert programmers were studied by Pennington (1987) [111] who found that experts use cross-referencing strategies whereas novices usually apply code-based and domain-based strategies. Davies (1993) [44] also studied coding activities by experts and novices in terms of information externalization strategies. He found that experts tend to rely much more upon the use of external memory sources.

2.1.5 Summary

Based on the above brief review of cognitive research in software engineering, we find that there are significant bodies of research on programming understanding and program debugging. However, there is not much research on incremental software development and program design. Particularly, there is not a common model proposed for all the software engineering processes, and all the existing models are one-dimensional and can only be applied to individual processes. On the other hand, no model explains the cognitive activities in detail, i.e. we do not know exactly how programmers do when they have conducted software engineering activities. Meanwhile, a well-defined cognitive model can also help to build tools which can improve the programming process and learning.

Although there is some research on expert programming, most of the research is not based on application by professionals. Also, there still exists a little

comparison between the cognitive activities taken by experts and novices on the same programming problems. Therefore, we have no very clear understanding of the differences between experts and novices from the cognitive point of view, even though we are convinced that a better understanding the differences between novices and experts can shed light on how to train novices into experts.

The generalization of previous research results were based primarily on procedural or declarative languages. The impact of OO paradigms on cognitive activities needs to be well studied.

The cognitive activities in software engineering can be studied from either individual or team perspective. However, most of the work done so far was based on individual programmers. There is no much Work on the collective and collaborative aspects of cognitive activities on some software processes. Some software engineering processes, such as pair programming, greatly involve the knowledge from a team of programmers, or a pair of programmers. The study of knowledge shared and cognitive activities should be useful to improve the processes.

The cognitive processes during program comprehension and program debugging need study from a different angle in order to understand the involved cognitive activities in detail.

2.2 Cognitive Research Methods

2.2.1 Think-Aloud Protocol

Cognitive scientists commonly use protocol analysis, task analysis, and discourse analysis [51, 58, 133] in their empirical work.

Task analysis is one of the techniques that study both the subject's actions and the accompanying cognitive processes [133] and which is commonly used to study subjects who have problems mastering complex behaviors. The purpose is to get an insight into the nature of a task when it is performed in the field. Task analysis addresses what is done, instead of what should be done, and it is seen as a useful tool for human-computer interaction research. The data is collected through the observations and the interviews with experts.

Discourse analysis is a way of approaching and thinking about a problem[58]. It is neither a qualitative nor a quantitative research method, but rather a manner of questioning the basic assumptions currently held. It allows access to the ontological and epistemological assumptions behind a project or a method of research.

Protocol analysis is a rigorous methodology for eliciting verbal reports of thought sequences [51] and is the most common method to study human cognitive processes. It identifies basic knowledge objects. In protocol analysis, subjects give verbal reports on the cognitive processes they are experiencing during a task. The three types of protocols involved in this are introspective, retrospective, and think-aloud [70]. Introspective and think-aloud protocols require the subjects to report their thoughts during a task, whereas retrospective protocols are gathered after a task is finished. Both introspective and retrospective reports require the subjects to

interpret the cognitive processes they use [51]. However in think-aloud protocol, subjects simply say what comes to their mind when they are completing a task; they are not supposed to comment on or to interpret their thoughts. All these types of verbal protocols require that one subject work individually.

The think-aloud method provides a way to collect data that could not be otherwise collected. It allows us to understand not only what the subjects are doing, but also why, and provides a basis for investigating the underlying mental processess during complex tasks.

The think-aloud protocol was used in 1920's by Watson [166] to illustrate general characteristics of the cognitive process in problem solving. The approach was later developed into a research method, especially when tape recorders became available in 1945. For example, Duncker (1945) [50] used it to analyze problem-solving processes in terms of memory search. The think-aloud method was systematically described by Newell and Simon (1972) [103] and used to build a detailed model of problem-solving processes.

Although the think-aloud protocol was originally used in psychology [51], it has now been used in other areas, such as medicine, usability evaluation [11], engineering design [65] and software comprehension [156], [158]. In most cases, think-aloud questions are used to stimulate verbalization, such as "what are you thinking now", "keep talking", "think aloud", and so forth.

Think-aloud protocol forces the subjects to speak out what is in their mind at a particular time. Although Ericsson and Simon (1993) [51] did not find changes in the cognitive process when subjects were asked to verbalize their thoughts, they

found that certain changes occured when subjects were asked to explain their cognitive processes.

A serious problem with think-aloud protocols is that subjects may endavour to deliberately produce the desired data. This effect is called “placebo effect” [127]. The outcome of an experiment may be biased if the experimenter unintentionally indicates the expectations by verbal communications or gestures like nodding when an expected result is observed [126].

The presence of an experimenter/mentor can also be problematic because it might affect the behavior of the subjects. Orne (1969) [108] stated that subjects can never be neutral to an experiment; the Hawthorne effect [1] will happen when subjects know that they are being studied, and that affects the outcome of the experiment. In particular, the think-aloud protocol with observation and the communication between mentor and subjects might affect the outcome [127].

A verbal report of the mental process may change the way in which a subject interacts with the task and this may affect the subject’s decision process. The think-aloud protocol forces subjects to speak and it interferes with the thinking process[92]. Some subjects may experience difficulties talking when they perform their tasks, or they may be affected by the presence of the mentor [51]. Some people may find it difficult to verbalize their thoughts [11].

Therefore, we need to search for additional empirical methods which can be used to study the cognitive process during software engineering processes in order to offer relief from those problems caused by the think-aloud protocol.

2.2.2 Recording Methods

The mental process of subjects requires verbalization and must be recorded for further analysis and research.

There are many ways to record the activities performed by subjects, such as audio recording, video recording, user notebooks, and computer logging. Among the recording methods, video and audio recordings are most commonly used. They provide rich and relatively permanent primary records.

Videotaping

Videotaping produces a continuous view of events and provides a detailed record of the process. Videotaping records the movements of the mouse, the code changes, and the communication between programmers. Normal observation tends to emphasize events that occur more frequently, since there is more data to be compared, whereas a video recording enables a repeated observation and exploration of a rare event [51].

Individual scenes can be replayed numerous times as the researchers reflect on what has occurred, and premature conclusions can be reduced or avoided [51]. Video provides continuous data rather than snapshot data and, hence, it records the richness of interactions. This is of great benefit. Data maintenance is also easy since we can store it in computer files, make copies without degradation, or copy them directly to CDs or DVDs. We also have many options for editing.

While videotaping offers many advantages, there are also limitations and weaknesses. Subjects may be more awkward or shy in front of a video camera than when they are near a portable audio recorder or microphone, since the later is less noticeable or conspicuous.

The playback of a digital video may be more complex for the experimenter than the playback of digital audio. Digital camcorders require a learning curve to successful operation. A great deal of work is involved when video data is transferred to a computer. This includes the file format, the video compression scheme, the output file types, the delivery medium etc. Producing the digital video file is also time-consuming. Rendering a full-screen video of less than an hour's duration on a 2GHz PC with 512 Mb of RAM can take up to 4-5 hours to complete.

The cost of equipment resources, the learning curve, and the production time for producing digital audio clips is less than the cost occurred in digital video production. Shooting, editing and producing video requires more time to learn and to produce than does audio production. Due to the huge amount of information being processed, video clips processing requires a higher level computer, while a low-end PC would perform the task satisfactorily in audio production.

Voice recording

As recently as ten years ago, voice-recording used to be commonly used for recording processes during cognitive research. Until the latter part of 1990s, most voice recording in experiment or field study had been done with the traditional

cassette tape recorder. Subsequently, the digital recording devices have been invented and voice recording software has been developed. They now clearly outrank tape recorders in terms of sound quality, reliability, and usability.

Table 2.1: The comparison between videotaping and voice-recording

	Advantages	**Disadvantages**
Videotaping	Can capture screen dynamically; Can capture everything we need; Easy to translate the data into episodes; Have high recording quality; Be easy to maintain and store the tapes	Higher cost of purchasing tapes and equipment; Have to have the mentor presented; Need to Change tape which could lost data; Have some Hawthorne effect; Editing takes time; Coordinate takes time; Need a learning curve to use the equipment.
Voice-recording	Cost effective with free software; Be flexible to schedule; Can capture screen-shots as needed; Less Hawthorne effect; Easy to edit the data	Cannot capture the whole process; Have a little more work to translate into episodes; Recording quality could be low.

Under certain circumstances, voice-recording is more preferred to video-taping, because of the limitation of videotaping we have discussed in the previous section. The advantages of voice-recording method are commonly accepted to be:

- No learning required to manipulate techniques
- Easy to conduct
- Easy to play back
- Minimum Hawthorne effect since the microphone can be put in a hidden place
- Less cost in equipment

Of course, voice-recording cannot record the screen changes made by programmers, and mouse movement. Therefore, at present, it is not commonly used.

Table 2.1 lists the comparison between videotaping and voice-recording.

2.2.3 Coding Schemes

After the data is captured, it is transcribed into “raw protocol”. The raw protocol is then divided into small units called “segments” or “episodes”. After the data is transcribed and divided into episodes, data analysis is conducted.

Some researchers, such as Meijer and Riemersma (1986) [98] and Confrey (1994) [35] have supported their conclusions by directly citing the protocols. However, the coding scheme is becoming more and more popular; without a coding scheme, an analysis on the data is difficult or sometimes even impossible to complete [59].

In software engineering, von Mayrhauser and Lang (1999) [156] pointed out that one of the difficulties for protocol analysis is to compare the results across different studies. That is why a coding scheme is needed to provide a unified classification; the scheme facilitates the replication of the protocol analysis and the assessment of the validity of its results.

Pennington (1987) [111] proposed a coding scheme for program comprehension analysis. Oslo et al. (1992) [107] developed a coding scheme to observe the programmers’ behavior during design meeting. Shaft (1995) [132]

developed a coding scheme to help code the verbal data when she studied the role of domain knowledge in program comprehension. Most recently, von Mayrhauser and Lang (1999) [156] described a detailed coding scheme for systematic analysis of program comprehension.

Therefore, additional code scheme or cognitive model may help to understand the cognitive process during software engineering processes and in turn, may improve the quality of software developed.

2.3 Cognitive Theories

2.3.1 Introduction

There have been numerous publications on constructivist learning, including the classical work of Piaget (1954) [114]. Piaget observed two learning activities: assimilation and accommodation. Von Glasersfeld's summary in [155] also describes these activities. In [106], Novak (1998) addressed theories of learning, knowledge, and instruction, and used concept maps as tools to describe the knowledge.

Many learning theories were developed for the purpose of providing an explanation and analysis of classroom learning [48]. Bloom and his colleagues [18] identified six levels of learning known as the Bloom taxonomy of the cognitive domain, starting from the knowledge (or recognition) at its lowest level, followed by comprehension, application, analysis, synthesis, and evaluation.

Vygotsky (1978) [161] asserted that culture is the most important issue in individual learning. His social cognition learning theory emphasizes social interaction in the development of cognition. Fischer and Ostwald (2001) [54] discussed the social and evolutionary process involved in knowledge construction during software development. Although social interaction is an important element during software engineering processes, we believe that our emphasis must be on individual programmer and the autonomous construction of their knowledge. As a result, we find this theory less applicable.

Distributed cognition was proposed by Hollan et al. (2000) [76]. It encompasses interactions among people, resources, and materials. Wallenstein (2003) [163] proposed a framework for a development tool design based on distributed cognition theory. Since some of the software engineering processes, such as program comprehension, program debugging, are often individual work that involves less resources and interaction with other people, therefore distributed cognition theory is not applicable here.

Observational learning [15] states that learning occurs through the simple processes of observing someone else's activities. However, software engineering processes involve more complicated processes than simple observation; therefore, observational learning is not suitable for studying the cognitive process during software engineering processes.

Gardner (1993) [64] proposed the theory of multiple intelligences and considered eight different intelligences to count for a broader range of human learning potential: linguistic, logical-mathematical, spatial, bodily-kinesthetic, musical,

interpersonal, intrapersonal, and naturalist. The theory provides guidance for an optimal use of talent, but it does not concentrate on the cognitive activities involved in the learning. Therefore we do not use it in explaining software engineering processes.

Brain-based learning is based on the structure and function of the human brain and emphasizes the fact that the brain can perform several activities spontaneously, such as the processes of tasting and smelling. Learning involves both focused attention and peripheral perception and requires both conscious and unconscious processes [82]. Because brain-based learning deals with the functions of the brain rather than the cognitive activity of learners, we did not find it applicable in our situation.

Control theory claims that behavior is not caused by a response to an outside stimulus, but inspired by what a person wants most, such as survival, love, and freedom [67]. Again we do not find that view useful for the study of software engineering processes.

Baragry and Reed (2001) [8] viewed software engineering processes using other disciplines such as metaphysics, epistemology, and psychology of cognition. To understand the design process, Ribbins et al. (1998) [123] used several cognition theories, including Reflection-In-Action. We believe that it might be useful to study the software engineering processes from disciplines other than the cognitive science, but the learning theory should be the more useful one.

We concluded that although these theories contain valuable insights, they are not directly applicable to our purposes in the study of the cognitive activities during incremental software development, program debugging and program comprehension.

The most promising theories we found include the constructivist learning theory and Bloom taxonomy [177]. We adopted them as components of our self-directed learning model and will explain them in the next two sections.

2.3.2 Constructivist Learning Theory

Constructivist learning theory explains the process through which people learn new facts and then add them to their knowledge base. It is based on the work of Piaget (1955) [114]. The original aim of Piaget's work was to explain learning in children, but the constructivist theory also extends to adult learning and to epistemology [155]. In [155], von Glaserfled (1995) described constructivism as a theory of knowledge. He defined radical constructivism as a theory of learning, and saw facts as being actively received either through the senses or through communication.

The basic assumption of this theory is the hypothesis that the learners actively and incrementally construct their knowledge. They start from some pre-existing knowledge and extend it by connecting new facts to previously learned knowledge. According to constructivism, it is not possible to assimilate new knowledge without having some structure developed from a previous knowledge. Knowledge is actively constructed by learners [60]. It is individually constructed by learners who try to explain things they do not completely understand and socially constructed by a group of learners who convey meanings to each other. They may

go through stages in which they may accept ideas that they will later discard as being wrong.

Piaget characterized constructivist learning in terms of two cognitive activities, assimilation and accommodation. Assimilation describes how learners deal with new knowledge, whereas accommodation describes how learners reorganize their existing knowledge. Assimilation and accommodation are complementary and inseparable, although one may be predominate at any particular time. They are the two poles of an interaction between the organism and the environment.

Constructivist learning theory explains the learning process for adults and children. Since the software engineering process can be viewed as a problem-solving process and a learning process, it might be a good idea to apply the constructivist learning theory in explaining the cognitive activity in software engineering. On the other hand, the software engineering process may be a good test-bed for constructivist learning theory, by further identifying the activities involved.

Recently, Rajlich (2002) [115] and Exton (2002) [52] applied constructivist learning theory to program comprehension. Rajlich (2002) [115] considered that programmers begin with the pre-existing knowledge and conduct program comprehension using two processes: assimilation and adaptation. Exton (2002) [52] overviewed the existing program comprehension strategies and correlated them with constructivism learning theory.

Rajlich and Xu (2003) [119] divided the two activities: assimilation and accommodation into four activities: absorption, denial, reorganization and expulsion. They provided a comparison between constructivist learning and the incremental software development process. Rajlich and Xu (2003) [119] also correlated the four programming activities: incremental change, reject of change request, refactoring and retraction, with the four cognitive activities. We used the extended constructivist theory as a component of our self-directed learning theory [177] [174].

2.3.3 Bloom's Taxonomy of the Cognitive Domain

Beginning in 1948, Bloom headed a group of educational psychologists who undertook the task of classifying educational goals and objectives. They [18] identified three domains of learning: cognitive, affective, and psychomotor. The cognitive domain is for mental skills, the affective domain is for growth in feelings or emotional areas, and the psychomotor domain is for manual or physical skills. The first one is commonly referred to as "Bloom's Taxonomy of the Cognitive Domain"[18]. For simplicity, we refer in this thesis to the Bloom's Taxonomy of the Cognitive Domain as Bloom's taxonomy or Bloom's levels.

Bloom identified six levels within the cognitive domain. The lowest level is the simple recall (knowledge) or recognition of facts. Then there are increasingly more complex (comprehension, application, and analysis) and abstract mental levels until the levels of synthesis and evaluation [18] are reached. The categories can be considered as degrees of difficulty and proficiency.

Table 2.2: The cognitive domain taxonomy and illustrative examples

Level	Sample verbs	Sample Behaviors
Recognition or Knowledge	collect, copy, define, describe, enumerate, examine, identify, label, list, name, quote, read, recall, retell, record, repeat, reproduce, select, state, tell	Student is able to define the six levels of Bloom's taxonomy
Comprehension	associate, cite, compare, contrast, convert, differentiate, discuss, distinguish, elaborate, estimate, explain, extend, generalize, give, group, illustrate, interact, interpret, observe, order, paraphrase, review, restate, rewrite, subtract, trace	Student explains the purpose of Bloom's taxonomy.
Application	administer, apply, calculate, capture, change, classify, complete, compute, construct, demonstrate, derive, determine, discover, draw, establish, experiment, illustrate, investigate, manipulate, modify, operate, practice, prepare, process, produce, protect, relate, report, show, simulate, solve, use	Student writes an instructional objective for each level of Bloom's taxonomy.
Analysis	analyze, arrange, breakdown, classify, compare, connect, contrast, correlate, detect, diagram, discriminate, distinguish, divide, explain, identify, illustrate, infer, layout, outline, points out, prioritize, select, separate, subdivide	Student compares and contrasts the cognitive and affective domains.
Synthesis	adapt, combine, compile, compose, construct, correspond, create, depict, design, devise, express, format, formulate, facilitate, improve, integrate, invent, plan, propose, rearrange, reconstruct, refer, relate, reorganize, revise, specify, speculate, substitute	Student designs a classification scheme for educational objectives that combines the cognitive, affective, and psychomotor domains.
Evaluation	appraise, assess, conclude, criticize, convince, decide, defend, discriminate, evaluate, explain, grade, judge, justify, measure, rank, recommend, reframe, support, test, validate, verify	Student judges the effectiveness of Bloom's taxonomy.

Table 2.2 outlines each level of cognition along with illustrating verbs and simple examples that illustrate the type of the thought activity of that level [42, 79]. Bloom's taxonomy is easily understood and has been studied and widely used

today in education. Teachers often use it to guide their instruction and to prepare questions for students [149]. There is some debate in educational circles if the synthesis layer is truly below the evaluation layer [104], but that discussion is beyond the scope of this thesis and does not impact our conclusions.

The six levels of Bloom's taxonomy clearly explain the mental process involved in student learning in the classroom. Software engineering process involves cognitive activities and is also a learning process. It would be useful to further study Bloom's Taxonomy of the Cognitive Domain, and to use it for analyzing the software engineering processes.

Bloom taxonomy has been applied by Buckley and Exton (2003) [25] to assess programmers' knowledge. They used Bloom taxonomy to correlate various software maintenance tasks.

We [173] used the Bloom taxonomy to study the program debugging process. We identified that programmers have to move from one level to another higher level of Bloom's taxonomy in order to make updates and debug a program.

CHAPTER 3

RESEARCH METHODS

Research methods used in this thesis are discussed in this chapter. We mainly applied case study method in our work. We also developed dialog-based protocol method for our research which is described below. In this chapter, we also discuss the recording methods used in our work.

3.1 Case Studies

Case study research is the most common qualitative method used in information systems [2]. Among numerous definitions of the scope of a case study, a typical one is as follows [178]:

A case study is an empirical inquiry that:

“Investigates a contemporary phenomenon within its real-life context, especially when the boundaries between phenomenon and context are not clearly evident."

Case study research can be positivist, interpretive, or critical, depending upon the underlying philosophical assumptions of the researcher. Correspondingly, there are three kinds of case studies: exploratory, explanatory, and descriptive.

The essential characteristic of case studies is that it strives towards a holistic understanding of systems of action. Therefore, case studies are a valuable method

of research, with distinctive characteristics that make them ideal for many types of investigations.

In order to study the cognitive process, we conducted a series of case studies on incremental software development, program debugging and software evolution after having obtained significant experience from the pilot study. The basic method was to give program problems and ask the programmers to conduct incremental software development or to evolve the programs. They were recorded in order to document all the information involved during this process. They were given a guideline to direct their experiment and monitored. Based on raw data, we did analysis and modeling, recognizing the cognitive activities with the help of our cognitive model.

For each case study on incremental software development and program debugging, we used our empirical method to collect the following data for individual action or mental activity as taken by programmers:

- Construction activities (absorption, reorganization, denial, expulsion)
- Bloom's taxonomy levels (recognition, comprehension, application, analysis, synthesis and evaluation)
- Domain concepts and programming concepts created or revisited
- Phenomena occurred and their relationship with concepts
- The number of times of the concepts which have been visited
- The number of new concepts added

Then we used the self-directed learning theory to explain our case study results. The self-directed learning theory is described in Chapter 4 in detail.

The pilot study is described in Appendix A. Other case studies are discussed in Chapters 5, 6, and 7. The detailed data are in appendixes B and C.

3.2 Dialog-Based Protocol

3.2.1 Description of the Dialog-Based Protocol

As mentioned earlier, the think-aloud protocol is commonly used in cognitive research during software engineering processes. Parnas (2003) [110] discussed the limitations of empirical studies in software engineering due to the Hawthorne effect[1]. In a previous chapter, we have discussed the think-aloud protocol, and have discovered some of its defects. In order to reduce the defects of think-aloud protocol, particularly the Hawthorne effect and placebo effect, we propose a dialog-based protocol (see Figure 3.1).

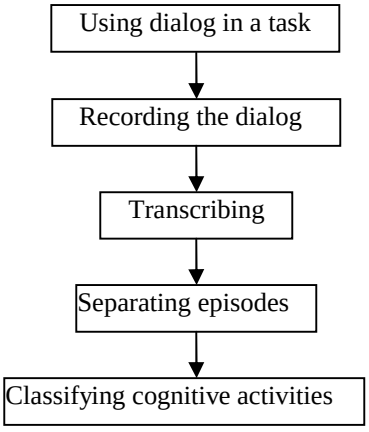

Figure 3.1: Steps of dialog-based protocol

The dialog is a conversation between two people, and it allows the two people to share ideas and to learn from each other. The dialog contains a large quantity of information which has been used as the raw data for studies in many fields, such as linguistics, psychology, sociology, and literature [19].

During pair programming, two programmers work side-by-side on design, algorithm selection, implementation, and testing at one machine [10]. Since the programmers need to cooperate with each other, they must verbalize their thoughts to each other so as to enable their partner to receive and understand those ideas. The dialog allows the mental process to be easily verbalized and thus, can reduce the Hawthorne effect and placebo effect, since programmers can easily forget that they are being studied when they talk freely.

If the paired programmers are allowed to speak of all that comes to their minds, one can record their communication and analyze it from the point of view of cognitive activities. We call this method the Dialog-Based Protocol. It is derived from the idea of pair programming [10]. The dialog-based protocol is similar to the think-aloud protocol. One major difference between the two protocols lies in the fact that in the think-aloud protocol only one subject is required to speak his/her own thoughts, whereas in the dialog-based protocol, two people are verbally expressing their thoughts while they complete their tasks. Both protocol techniques require the process to be recorded for analysis, either as audio or video. The anticipated advantages are summarized in Table 3.1 and discussed in the following section when a comparison is made with think-aloud protocol.

3.2.2 Comparison with the Think-Aloud Protocol

According to Ericsson and Simon (1993) [51], the think-aloud protocol can only produce a subset of thoughts. We speculate that dialog-based protocol can force partners to explain all the essential issues clearly to each other, thus, allowing for a more complete documentation of the process. Also, the dialog-based protocol requires continuous verbalization during a programming task. This could provide a more complete picture of the cognitive process than the think-aloud protocol, because the latter is very likely to be constantly disrupted by the experimenter. However, since subconscious thoughts will be missing from the dialog, we realize that even in dialog-based settings the recorded dialogs might not be complete.

Correctness of the data is also an important factor in protocol analysis. Because of the disturbance of the subjects in the experiment, the think-aloud data might contain invalid portions. Questioning or reminding the subjects may change the cognitive processes. In the dialog-based setting, there is no communication between the mentor and the subjects. Therefore, we speculate that the invalid portion of data is reduced.

Another problem associated with the think-aloud protocol is the misinterpretation, because some subjects may have difficulties in explaining the process. However, in dialog-based sessions, subjects have to explain all important points exactly to their partners, and as a result, such misinterpretations should be reduced.

Table 3.1: Anticipated comparisons between Think-Aloud Protocol and Dialog-Based Protocol

	Think-Aloud	**Dialog-Based**
Completeness of data	incomplete	more complete
Correctness of data	moderate	high
Hawthorne effect	present	minimized
Placebo effect	present	minimized
Performance of subjects	more affected	less affected
Mentor Presence	necessary	unnecessary
Pairing problem	does not exist	exists

The think-aloud protocol forces subjects to speak and interferes with the thinking process [92]. Some subjects may experience difficulties talking when they perform their tasks, or they may be affected by the presence of the mentor [51]. Some people may find it difficult to verbalize their every thought [11].

We speculate that dialogs can be recorded under more natural circumstances than think-aloud sessions. When people collaborate, they are required to give arguments in order to clarify their thinking. They might forget that they are recorded, thereby reducing the Hawthorne effect.

The think-aloud protocol may also impact the programmers' performance, because they have to conduct the tasks and also verbalize them. Berry and Broadbent (1990) [14] reported that people who were not thinking aloud performed faster than those who were. Since dialogue is inherently a part of many activities involving partners, the dialog-based protocol incurs no performance overhead when used in an applicable experiment (see Table 3.1).

The mentor must be present in think-aloud protocol since subjects need to be reminded to verbalize the thinking. For the dialog-based protocol, the presence of the mentor is not necessary since the programmers need to communicate anyway (see Table 3.1). Therefore, the placebo effect is reduced. Besides, the cost of the presence of the mentor during the experiment is higher in think-aloud protocol than in dialog-based protocol.

Pairing might be an issue in the dialog-based protocol; some pairs work well together while others do not. Some subjects may be embarrassed to speak out in a pair whereas he/she would be more willing to comment in a think-aloud setting. Therefore, the subjects chosen for the case study have to be good communicators and two programmers in the pair should be able to work together.

3.3 Recording Methods

The recording methods might affect the experiment as well, because when the subjects know that they are being recorded they might alter their behavior. Therefore, choosing the right recording method for data collecting is an important issue. We used videotaping and screen capturing in our experiment.

3.3.1 Videotaping

As discussed in Chapter 2, videotaping is the most common recording method used for collecting empirical data in cognitive research. The video provides

continuous data rather than snapshot data, and it records the richness of interactions. We used digital camera to record two pairs' experiment.

3.3.2 Software Screen-Capturing

For the study of cognitive activities in software engineering, the screen changes performed by the programmers and their communication are far more important than other visual information, such as programmer movements. Screen-capturing software records screen changes.

In recent years, the screen-capturing programs have become more sophisticated. The software not only captures the screen changes, but also records the communication between programmers. The screen-capturing software has evolved into an effective tool for collecting data during cognitive research.

Screen-capturing does not require extensive operation knowledge, since it creates media files which can be operated easily. There can be a few seconds' delay in the screen change capture. As a result, there is a possibility that some faster changes may have happened during this time and that they are not found in the screenshots. However, since programmers can not change code as fast, such delays do not produce any substantial loss of data.

An example of screen-capturing software is Microsoft Producer [99], which supports screen-capturing and voice-recording automatically at the same time. With Microsoft Producer, the mouse events, the communication between the two programmers, and the changes of the screen are digitally recorded. Although it

does not record programmers themselves, Microsoft Producer captures all of the essential data during the programming process. Microsoft Producer delivers high quality audio and video files in various formats. In this way, the programming process and its environment can be stored or replayed for further analysis.

We used Microsoft Producer to record the programming activities for eight pairs when we conducted experiment on incremental software development.

CHAPTER 4

SELF-DIRECTED LEARNING THEORY

In Chapter 2, we have briefly reviewed the code schemes used by researchers. In this chapter we will describe our novel model: Self-Directed Learning Theory, which can be used as a coding scheme to analyze the cognitive activity during incremental software development, program debugging and software evolution [177].

Before applying the code scheme to do any analysis, the raw protocol should be divided into “segments” or “episodes”. Some researchers divided the raw protocols into episodes based on the subject’s intentions and actions [65], while others used verbalization events or syntactic markers [51].

In our case study, we divided the dialog into episodes based on concepts that the dialog deals with. We classified the concepts as domain concepts and programming concepts. Domain concepts belong to a specific application domain of the program [16], such as strike and spare in the bowling application, whereas programming concepts belong to the knowledge of programming, such as the programming language, program development process, design decisions, and others. Since they drive the software development, we considered only the design decisions in programming concepts, while other programming concepts only play a background role. We used the following rule to separate dialog into episodes: a new episode begins when programmers start the discussion of a different concept. If several concepts are discussed at the same time, a new episode starts when one of

the concepts is dropped. When no significant concept is involved in a particular part of the dialog, we combined this part with the previous episode.

4.1 Description of the Self-Directed Learning Theory

The self-directed learning theory extends the constructivist learning theory [114] and combines it with the Bloom taxonomy of the cognitive domain [18]. The extended constructivist learning theory is used to define the differences among the cognitive activities and Bloom's taxonomy is used to classify the cognitive levels.

Table 4.1: The four cognitive activities and corresponding verbs

Activities	Sample verbs
Absorption	add, believe, choose, conclude, confirm, consider, create, define, demonstrate, determine, identify, image, imply, interpret, make out, prove, recognize, set up, show, start, think, verify, visualize
Denial	decline, disapprove, refuse, reject, turn down
Reorganization	adjust, alter, break, change, extract, fix, modify, move, pull out, refactor, regroup, tune up
Expulsion	delete, dismiss, eliminate, erase, exclude, expel, force out, get rid of, kill, remove, take out, throw out, withdraw

As we mentioned in Chapter 2, constructivist learning is based on the work of Piaget on child learning [114]. Piaget identified two learning activities: assimilation and accommodation. Assimilation describes how learners deal with new knowledge, whereas accommodation describes how learners reorganize their existing knowledge.

We [119] find that assimilation can be subdivided into two separated activities: absorption and denial. Absorption refers to the case when learners add

new facts to their knowledge, and denial occurs when learners reject the new facts because they do not fit into the existing knowledge base. Accommodation was also divided into two separate activities. When the learners reorganize their knowledge and apply it to aid future absorption of new facts, we call their cognitive activity reorganization. When a part of the knowledge becomes obsolete and the learners reject it, the corresponding activity is called expulsion. Table 4.1 contains the four cognitive activities and the characteristic verbs that programmers use when using the specific activity.

In order for learning to occur, the learners must possess preliminary knowledge and they must be in a learning environment. Preliminary knowledge or capabilities make learning possible; the more the learners know, the more they can learn. We find that learners' preliminary knowledge is based on the experience that was supplemented with the obvious and easy-to-find facts about the program during software engineering. Sometimes this preliminary knowledge turned out to be inaccurate or even completely wrong, and the learners employed the four cognitive activities to build more accurate knowledge. The result of learning is resultant knowledge.

Bloom and his colleagues identified six levels of learning known as the Bloom taxonomy of the cognitive domain; see Table 2.2. The six levels include the knowledge or recognition of facts at its lowest level, followed by comprehension, application, analysis, synthesis, and evaluation [18].

Bloom taxonomy is a well-established part of the theory of learning and we applied it to classify the levels of the programmer cognitive activities. The Bloom

levels are also assigned to each episode through the use of characteristic verbs (Table 2.2). Table 2.2 is a composite of similar tables in [42, 79] and it was cleaned for the purposes of software engineering, where some verbs attained a different meaning, for example the verb compile.

Table 4.2: The self-directed learning model: All four cognitive activities may take place at any of the Bloom levels

	Assimilation		Accommodation	
	Absorption	**Denial**	**Reorganization**	**Expulsion**
Recognition	v	v	v	v
Comprehension	v	v	v	v
Application	v	v	v	v
Analysis	v	v	v	v
Synthesis	v	v	v	v
Evaluation	v	v	v	v

The self-directed learning model is a two dimensional model, composed in one dimension of the four cognitive activities and on the other dimension of six levels of Bloom taxonomy (see Table 4.2). The theory is particularly suitable to the situations where learners must discover the facts of the knowledge on their own, without a teacher. This is a common situation in software engineering.

Once the dialog has been decomposed into episodes, the self-directed learning theory is used for protocol analysis. Each episode is classified by using the characteristic verbs from Table 4.1 and Table 2.2. The final data is computed as the frequencies of occurrences in episodes for each activity type and for each Bloom level.

We have applied the self-directed learning theory and dialog-based protocol[174] [177] to conduct case studies on incremental software development and in program debugging [173].

4.2 Comparison with Other Models

Over the past twenty years, there have been many cognitive models proposed for program comprehension. The models include top-down model, bottom-up model and as-needed model [24] [90] [91].

As discussed in Chapter 2, we find that all those existing models (top-down, bottom-up etc) agree that the program comprehension process uses existing knowledge combined with a comprehension strategy in order to acquire new knowledge. This means that both assimilation and accommodation occur during program comprehension, which coincides with our self-directed learning model. However, there are some significant defects in those previous models.

First, none of them describes in detail the actual cognitive activities. All models describe the simple procedure for program comprehension. For example, the bottom-up model only mentions that programmers combine together small chunks of source code to build up higher levels of knowledge, but it does not specify what kinds of activities programmers perform. The self-directed learning model classifies the cognitive process in detail into four activities at six Bloom's levels.

Secondly, according to the work of Pennington (1987) [111], and Von Mayrhauser and Vans (1998) [160], none of the existing models can explain the entire program comprehension process. Their claim that programmers would sometimes use the top-down approach, and sometimes use the bottom-up approach seems to be unreasonable. Our learning model is more complete and in greater detail. By integrating both top-down and bottom-up models it explains all the program comprehension processes (see Figure 4.1).

Thirdly, it seems that most existing models themselves cannot explain well the differences between experts and novices. In order to do so, several models such as bottom-up and top-down, have to be used. This again, appears unreasonable and inappropriate. For example, Burkhardt et al. (1998) [26] found out that experts often used top-down, while novices rarely used the top-down approach, but instead using bottom-up approach.

The efficiency of program understanding relies on several factors, but the programmer's experience in program comprehension is as crucial as his/her domain knowledge and relevant programming knowledge. The experience and the levels of domain knowledge obtained affect the process of creating and testing hypothesis. A programmer who is lacking in knowledge of the program's domain may not be able to synthesize and understand the code. This supports Brooks' emphasis on the importance of domain knowledge in the understanding process [24]. Consequently, our model can clearly explain the gap between experts and novices, one which seems a difficult task for some existing models.

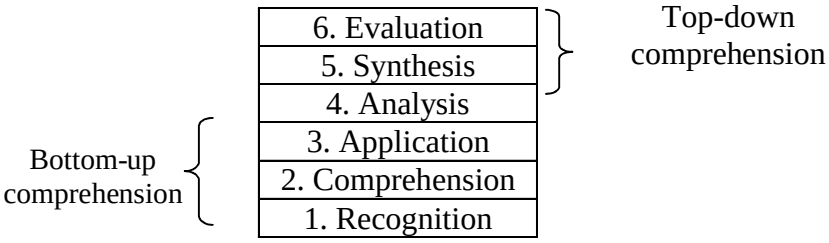

Figure 4.1: Comprehension strategies and their relationship with Bloom's levels

In summary, compared with our model, the top-down model mainly emphasizes synthesis, hypothesis and evaluation, only three of the Bloom's levels. The bottom-up model strengthens the knowledge, comprehension and application - the three lower levels of Bloom's taxonomy (see Figure 4.1).

The programmer is able to obtain and to understand the knowledge so as to synthesize information and to generate hypotheses, i.e. moving from lower levels of Bloom's levels to higher levels to form the understanding process. It is not possible for the programmer to do so without knowledge, which coincides with the work done by Soloway et al (1988) [142]. Soloway et al (1988) [142] found that the process of programming understanding is a cycle that includes reading the code (obtaining the knowledge), raising questions (apply knowledge), making a conjecture (synthesize and generate hypothesis) and searching code for answers (evaluation). The process is also called the read-question-conjecture-search cycle, which well fit in with the learning levels in our learning model.

CHAPTER 5

CASE STUDY ON INCREMENTAL SOFTWARE DEVELOPMENT

5.1 Introduction

Incremental software development and subsequent program evolution embraces change request analysis, impact analysis, design, coding, debugging and testing. Programmers often start with an existing program or a program framework that covers fundamental functionality, and then add and delete properties to and from it. Thus, the cognitive process is central and important. Since the programmers at different levels might learn differently [141], finding out the differences in terms of cognitive activities is an essential part of these studies.

In this work, we used our new empirical approach including the dialog-based protocol, software screen-capturing, and the self-directed learning theory to study programmer learning and cognitive process during the development of a bowling program that is a part of the Martin and Koss' case study [93]. We replicated this program development of ten intermediate level pairs using test-first driven and refactoring techniques. Pair programming entails a conversation at many levels (change request analysis, design, coding, debugging and testing), thus providing an opportunity to analyze what that knowledge is consisted of and what cognitive processes are involved. The test-first technique [10] requires that programmers define the unit tests before writing the unit code. It forces the programmers to define the exact functionality of each method which means that the system will be

automatically tested as it is developed. Refactoring transforms the source code so that it is more readable and easier to update [10]. We paid particular attention to the differences between theirs and the experts' learning and the cognitive activities that were involved. The purpose of the case study is to either falsify or to prove the hypotheses which were stated in Chapter 1 and are repeated here:

1 "There are some better ways than think-aloud protocol to study the cognitive activities in software engineering"
2 "The combination of learning theories can be used to better study the cognitive activities during software engineering process"
3 "Experts and intermediate programmers do follow the same cognitive processes when they perform incremental software development"
4 "Intermediate programmers follow a similar cognitive process during incremental software development"

Since we used a new empirical technique, we also carefully observed the strengths and weaknesses of this technique which is discussed in Chapter 8 in detail.

5.2 Case Study Design

We conducted case study on ten pairs. Partial results were published in [174] and [177] [176]. The dialogue of Martin and Koss was recorded and a reenactment of that pair programming episode was published in [93].

5.2.1 Participants

Table 5.1: Programmers' background and the date of experiment

Pair	Level	Month
Pair I	Graduate	March 2004
Pair II	Graduate	March 2004
Pair III	Graduate	February 2005
Pair IV	Graduate	February 2005
Pair V	Graduate	February 2005
Pair VI	Graduate	February 2005
Pair VII	Undergraduate	March 2005
Pair VIII	Undergraduate	March 2005

All participants are from the Department of Computer Science, Wayne State University. In March 2004, four graduate students were randomly selected from the graduate class CSC 7110 (Software Engineering Environments) which had 22 students in total. In February, 2005, all 12 graduate students of the CSC 7110 course were asked to take part in the case study. Four advanced undergraduate students were selected from a class of 24 in CSC 4110 – Software Engineering in March, 2005. All participants were classified as intermediate level programmers as according to their programming ability. They had programming experience with C, C++, and Java, but they had never used pair programming, test-first, or refactoring practices.

The data from two pairs were rejected as invalid, and this will be further discussed in Chapter 8. Table 5.1 shows the basic information of the remaining eight pairs of programmers whose data are valid. The case study compared the

results of these eight pairs to earlier published results of the expert programmers Martin and Koss [93].

5.2.2 Material

The task to be solved in the case study is to implement an application which records the Bowling scores for a bowling game. It requires the programmer pair to understand both domain concepts and programming concepts. We can make a comparison between the data collected during our case study and the original work done by Martin and Koss [93]. Martin and Koss' work was treated as the experts' work, since they had over 15 years of programming experience and a considerable background in pair programming, test-first, and refactoring practices. All the participants worked on the same task. The intermediate programmers were not originally familiar with the bowling domain, nor did they know the solution of Martin and Koss.

The simple requirements for the bowling application are described verbally as follows:

"To write an application that keeps track of a bowling league. The program needs to record all the games, and accurately score of each game, with the following simple bowling rules:

- Bowling is played on a lane with 10 pins at the end that the player should knock down with a ball he/she throws and makes rolling on the lane.
- The placing of the ten pins makes up what we call a frame.

- The object of the game is to knock down as many pins as the player can.
- A game is scored on a single line on the score sheet. The line is divided into 10 boxes or "frames".
- Scoring of the game proceeds from left to right, with the running total of pins knocked down noted in each frame.
- The player starts bowling each frame with all 10 pins standing. The player has two "throws" with aims to knock down all 10 pins for the current frame.
- If the player knocks down all the pins on the first throw, it is called a "strike". The number of pins knocked down by the next two throws is added as "bonus" points for this frame.
- If the player knocks down all the pins on the second throw, it's called a "spare". The number of pins knocked down with the next throw will be added as "bonus" points for this frame.
- If the player does not knock down all 10 pins in a frame after two throws, the number of pins knocked down is added to the running total marked in the frame.
- This counting may demand one or two extra throws to complete the score of the 10^{th} frame (one extra throw in case of spare and two in case of strike). Extra throws means extra frames (two if the 10^{th} frame is strike and the first extra throw is a strike too)".

Figure 5.1 shows how the bowling score is recorded. Each frame except the tenth has one little square in the upper right, where scoring for the frame's own throws is kept. The first throw is written outside the little square, the second is recorded inside the little square, and the cumulative score goes in the big lower part

of the square. If it is a “spare”, then the second throw is marked with dark triangle. If it is a “strike”, then a cross is marked in the second throw.

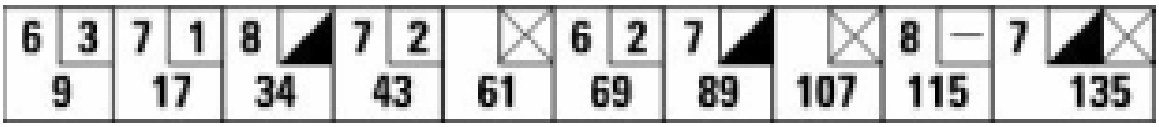

Figure 5.1: An example of scoring bowling games

5.2.3 Procedures

As mentioned earlier, the case study was carried out in March 2004, February 2005, and March 2005.

Since the programmers were new to dialog-based protocol, test-first programming and refactoring and it is particularly difficult for novices to adopt the test-first [101], the programmers were provided with a training session and reading materials on pair programming, test-first, and refactoring techniques prior to the case study. As a part of the training, they were asked to write a simple program using the Eclipse environment and JUnit [63] in order to understand the procedure and to familiarize themselves with those techniques and corresponding tools.

The case study was conducted in software engineering laboratory of Wayne State University. We videotaped the first two pairs. For the rest of pairs, we used a free software “Microsoft Producer” that allowed us to capture computer screens and record voice at the same time. The author acted as the mentor and monitored the process, recorded the data, and provided the programmers with a description of the

bowling rules. The mentor also answered the technical questions related to Eclipse compiler and JUnit testing tool [63], but did not give any guidance on the design and implementation of the program. The programmers were advised to speak loudly and communicate effectively.

The case study was divided into several sessions, each of which lasted about two hours.

Once the recording sessions were finished, recordings were transcribed and then analyzed. The dialogue was decomposed into episodes using the method described in Chapter 4.

Self-directed learning theory was employed for analyzing this observational data. The first analysis of the protocols involved assignment of activity types as specified by the self-directed learning model. We classified each episode according to the four cognitive activities (i.e. absorption, denial, reorganization, and expulsion) by using characteristic verbs defined in the self-directed learning model [174].

As a result of these activities, concepts were added or excluded from the existing knowledge. For example, during absorption, new concepts are added to the knowledge; during expulsion, concepts are excluded from the knowledge. The number of concepts in the knowledge was recorded for each episode.

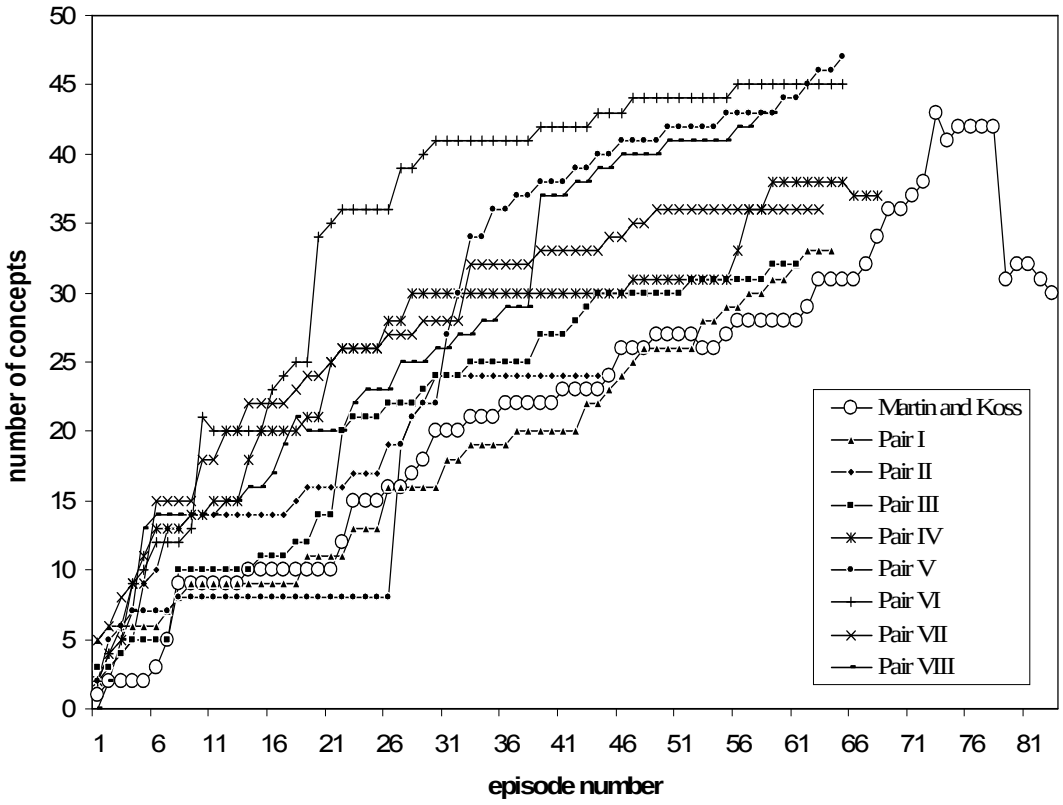

Figure 5.2: The number of concepts in the knowledge of each pair during incremental software development

Then, the Bloom's levels (i.e. recognition, comprehension, application, analysis, synthesis, and evaluation) were assigned to each episode also through the use of characteristic verbs (see Table 2.2). The final data was then computed as the frequencies of occurrences in episodes for each activity type and for each Bloom's level.

5.3 Case Study Result

The intermediate pairs took from two and half to five hours to complete the

task. All programs were completed and covered all significant test cases; the size ranged from two to six classes with a total of 19 to 28 methods.

Table 5.2: Main characteristics of the programs and the dialogs in the case study

	Martin and Koss	**Pair I**	**Pair II**	**Pair III**	**Pair IV**	**Pair V**	**Pair VI**	**Pair VII**	**Pair VIII**	**Average intermediate pairs**	**DD for intermediate pairs**
Lines of code	215	246	235	222	232	230	240	360	249	251.7	44.6
Number of class members	36	26	19	25	23	27	26	24	28	24.7	2.8
Number of classes	3	2	2	3	4	2	6	4	4	3.3	1.4
Number of domain concepts discussed before coding started	9	2	4	3	5	6	2	6	2	3.7	1.8
Maximum number of design decisions	43	33	24	32	38	47	45	36	43	37.2	7.7
Number of programming concepts the end of the task	30	33	24	32	37	47	45	36	43	37.1	7.7
Number of refactorings	16	5	2	9	3	2	3	0	2	3.2	2.7
Number of test cases	10	10	8	9	10	8	10	10	10	9.2	0.9
Number of episodes	84	65	45	62	69	67	66	64	60	62.3	7.5
Number episodes before coding	3	1	4	1	3	6	1	2	1	2.3	1.8
Time used (minutes)	X	288	330	146	307	294	295	154	245	257.4	70.3

Figure 5.2 shows the changing numbers of programming concepts in the knowledge. Table 5.2 summarizes the characteristics of the developed programs and of their respective dialogues. The final UML class diagrams [56] of the Martin and Koss' program and our pairs are shown in Appendix A and Appendix B. Table 5.3 and Table 5.4 show the detailed distribution of four cognitive activities at

different Bloom levels from the dialogue of Martin and Koss, and the cumulative data from the dialogues of all eight intermediate pairs, respectively. Table 5.5 and Table 5.6 present the data for individual pairs. The detail data for each pair is shown in Appendix B.

Table 5.3: The distribution of the cognitive activities and Bloom levels throughout the recorded episodes for Martin and Koss

	Assimilation		Accommodation		Total
	Absorption	Denial	Reorganization	Expulsion	
Recognition	0	0	0	0	0
Comprehension	20 23.8%	2 2.4%	5 5.9%	1 1.2%	28 33.3%
Application	9 10.8%	2 2.4%	5 5.9%	2 2.4%	18 21.5%
Analysis	17 20.2%	1 1.2%	5 5.9%	3 3.6%	26 30.9%
Synthesis	11 13.1%	0 0.00%	1 1.2%	0 0.0%	12 14.3%
Evaluation	0	0	0	0	0
Total	57 67.9%	5 6.0%	16 18.9%	6 7.2%	84 100%

Observations of Programming Process

Martin and Koss' program has a better design and their class members are much more elegant and readable than the classes produced by intermediate programmers. There is also a Scorer class separated from the class Game to calculate the scores for different cases such as spare, strike and normal throws, while none of the intermediate pairs has similar design. For example, Pair I used three arrays to store the scores, which turned out to be less efficient. The program implemented by pair II was hard to read with less meaningful variable names, and

other pairs had similar design problems. Experts implemented the program with 36 class members including instance variables and methods, while intermediate programmers had from 19 to 28 class members. The experts' program contained fewer numbers of lines of code (215) than those by intermediate programmer which ranged from 230 to 360.

Table 5.4: The distribution of the cognitive activities and Bloom levels throughout the recorded episodes for the eight intermediate programmer pairs

	Assimilation		Accommodation		Total
	Absorption	Denial	Reorganization	Expulsion	
Recognition	0	0	0	0	0
Comprehension	120 24.1%	1 0.2%	4 0.8%	2 0.4%	127 25.5%
Application	171 34.3%	1 0.2%	16 3.2%	5 1.0%	193 38.7%
Analysis	77 15.4%	0 0.0%	6 1.2%	3 0.6%	86 17.2%
Synthesis	93 18.6%	0 0.0%	0 0.0%	0 0.0%	93 18.6%
Evaluation	0	0	0	0	0
Total	461 92.4%	2 0.4%	26 5.2%	10 2.0%	499 100%

All intermediate pairs kept every single class they created until the end. One programmer pair created two classes at the beginning and kept them to the end, while another pair defined two classes at the beginning and created one new class later. However, experts created five classes at the beginning but only kept two of them to the end. A big drop in the numbers of valid design decisions in the Martin and Koss' work is indicated in Figure 5.2. This indicates that the expert programmers more readily abandoned obsolete or inadequate concepts, while the

intermediate programmers tried to fit the new design decisions into their previous design decisions and were reluctant to change them.

Table 5.5: The distribution of the cognitive activities for individual pairs

	Pair I	Pair II	Pair III	Pair IV	Pair V	Pair VI	Pair VII	Pair VIII	Average	SD
Absorption	57 87.7 %	42 93.3 %	52 83.9 %	65 94.2 %	64 95.5 %	61 92.4 %	63 98.4 %	57 95.0 %	57.6 92.5%	7.7
Denial	0 0%	1 2.2%	1 1.6%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0.25 0.4%	0.5
Reorganization	5 7.7%	2 4.5%	9 14.5 %	3 4.4%	2 3.0%	3 4.6%	0 0.0%	2 3.3%	3.25 5.3%	2.7
Expulsion	3 4.6%	0 0.0%	0 0.0%	1 1.4%	1 1.5%	2 3.0%	1 1.6%	1 1.7%	1.12 1.8%	1.0
Total	65	45	62	69	67	66	64	60	62.3	7.5

There were several sudden increases of numbers of concepts in the work of intermediate pairs as shown in Figure 5.2. This was caused by the fact that the intermediate programmers conducted a discussion of several concepts in one episode, while the experts referred to one concept at a time.

The discourse in the Martin and Koss dialogue was more coherent and richer than the dialogues of intermediate pairs since Martin and Koss' dialogue was edited from the original recorded data and phrasing is more mature and better articulated. Intermediate pairs used a lot of references to the source code, which were missing in the Martin and Koss dialogue.

The intermediate programmers applied refactoring several times at the beginning by renaming the variables, but they did not use more sophisticated refactorings like methods or class extraction [57]. Experts refactored the code when

the application was close to completion and they performed more advanced refactorings such as reducing the scope of variables and extracting methods. They also reduced the number of concepts towards the end of the process as a "final cleanup". As a result, their program was more elegant and comprehensible. However, as indicated by Figure 5.2, none of the intermediate pairs did the same thing.

Table 5.1 shows that both expert and intermediate programmers created a similar number of test cases. This indicates the intermediate programmers understood the test-first technique well and applied it as effectively as did the experts.

Observations of Cognitive Activities

Table 5.6: The distribution of the Bloom levels for individual pairs

	Pair I	Pair II	Pair III	Pair IV	Pair V	Pair VI	Pair VII	Pair VIII	Average	SD
Recognition	0	0	0	0	0	0	0	0	0	0
Comprehension	20 30.7%	14 31.1%	13 21.0%	16 23.2%	16 23.9%	13 19.7%	19 29.7%	15 25.3%	15.7 25.3%	2.6
Application	23 35.4%	20 44.5%	22 35.5%	28 40.6%	26 38.8%	25 37.9%	25 39.1%	25 41.7%	24.3 38.7%	2.5
Analysis	10 15.4%	6 13.3%	15 24.2%	11 15.9%	14 20.9%	14 21.2%	7 10.9%	8 13.3%	10.6 17.2%	3.5
Synthesis	12 18.5%	5 11.1%	12 19.3%	14 20.3%	11 16.4%	14 21.2%	13 20.3%	12 20.0%	11.6 18.8%	2.9
Evaluation	0	0	0	0	0	0	0	0	0	0
Total	65	45	62	69	67	66	64	60	62.2	7.5

All intermediate pairs started the dialog with a discussion of two to six domain concepts in about two episodes before coding started. In comparison,

expert programmers discussed nine domain concepts in three episodes before the coding started. Intermediate programmers discussed the domain concepts and programming concepts alternately at the beginning, while experts concentrated the discussion on the domain concepts. The fact that there are 84 episodes in the expert's work as compared to only 45 to 69 episodes in the intermediate programmers' work also indicates that the experts tend to switch concepts during the discussion more frequently than do intermediate programmers.

In the case study, the absorption activity dominated the programming process from the beginning to the end. This coincides with the fact that the most common goal of programmers during incremental software development is to create programming concepts that result in application classes and methods. For the intermediate programmers, absorption ranged from 84% to 98% of all activities with an average of 92.4%, while there was 67.87% in the Martin and Koss' work (see Table 5.4). This could indicate that compared to experts, the intermediate programmers have to absorb more knowledge in order to complete the programming task.

Most absorption activities occur at the lower Bloom levels such as comprehension and application, particularly for the intermediate programmers, who spent more than 58% of total activities on absorption at comprehension and application levels. Experts spent only 34% absorption activity at lower levels of Bloom taxonomy. Reorganization is the second most common activity because it is closely related to refactoring, and refactoring was a required part of the development process. Experts spent 19% of their efforts in reorganization activity,

while intermediate programmers had three episodes (5% of total) classified as reorganization (see Table 5.5).

Only a few episodes were classified as denial or expulsion, indicating that retraction of concepts from the program and/or rejection of the change requests seldom happens. Experts did expulsion in 7% of the episodes, pairs II and III did no expulsion at all, while pair I did expulsion in three episodes. The remaining pairs did expulsion in only one or two episodes. On average, the intermediate programmers did expulsion in only 1.8% of the episodes.

Of all episodes, about 64% were classified at the lower Bloom levels (recognition, comprehension, and application) in the dialogues of intermediate pairs, as compared to 54.7% of Martin and Koss (see Table 5.6 and Table 5.3). This indicates that intermediate programmers spent more time in understanding and applying the knowledge while experts took part of the knowledge as granted. On the other hand, experts spent more than 45% activities at higher Bloom levels (analysis, synthesis, and evaluation), compared to only 36% of intermediate programmers. It seems that experts know better how to analyze the knowledge, how to apply strategies to use the knowledge and to generate test cases.

These results, together with the comparison of the numbers of absorption activities, and the numbers of episodes classified at lower or higher levels of Bloom taxonomy in our replicated case study, coincide with Gilmore's observations [66] that novices and intermediates often take longer time to accumulate their knowledge than experts and even if they have as much knowledge as experts, they

may still need a longer time to finish their programming tasks, as they are not familiar with the strategies to use the knowledge.

No episodes were classified at the top (i.e. evaluation) or at the bottom (i.e. recognition) of Bloom taxonomy in our case study. This is different from a case study on program debugging [173] (see next Chapter), where we identified all six levels of Bloom taxonomy.

Recognition is the pre-requisite for the activities at comprehension and other higher levels. It is the main reason that it does not explicitly appear in our data. An additional reason is due to the way we split the dialogue into episodes.

The absence of evaluation episode in the case study may indicate the differences in the cognitive difficulties among different software engineering processes. We believe that the programming activity in this case study was quite simple and the size of the domain and the resulting software were small. During more complex software engineering tasks (i.e. debugging, reverse engineering, reengineering, etc.) more hypothesis-driven activities are performed, which would be classified at the top level in Bloom's taxonomy (i.e. evaluation), which has been found in our case study on program debugging [173].

Although many differences exist between the results of the experts and intermediate programmers, there are some similarities as well, such as the numbers of test cases created, and the numbers of class members. We believe that these numbers reflect the nature of the completed application rather than the expertise and therefore they were the same for both experts and the intermediate programmers.

5.4 Threats to Validity of the Case Study

This section lists threats to the validity of the case study [178], which could affect the generality and utility of the conclusions.

We used twenty graduate and advanced undergraduate students as subjects. Although students have a similar technical ability as intermediate level industry professionals, they might perform the tasks differently. A larger number of subjects might impact the results.

The problem solved by the programmers was relatively small. All the programmer pairs spent only two and half to five hours. A more complex problem or a problem in a different domain may produce different results.

The episodes were separated and classified manually; different authors may separate and classify episodes differently and a different separation criterion might yield different results.

Other threats associated with the empirical techniques employed are discussed in Chapter 8 in more detail.

5.5 Summary of the Case Study

In order to study cognitive process and programmer learning, we developed an empirical method that includes dialog-based protocol, software screen capturing and self-directed learning theory. Using this new approach, we conducted a case

study of expert and intermediate programmers during incremental software development in order to validate our novel approach and to study the cognitive process involved.

Our case study showed that the self-directed learning model can be used to study the cognitive activities during incremental software development. Our experience in experiment shows that dialog-based protocol is an exciting approach and reduces the placebo effect [127] and Hawthorne effect [1]. We also found that the Martin and Koss exercise is duplicable with different types of programmers. All intermediate level programmers followed a similar cognitive process with a similar distribution of the cognitive activities and Bloom levels.

Using incremental software development with the test-first approach, absorption is the dominant activity behind the process. Reorganization often occurs, but denial and expulsion occasionally appear. Four out of six of Bloom's levels were identified in the case study.

We also found that experts discussed related domain concepts more extensively before the coding started. The intermediate programmers often discussed several concepts at the same time, while experts mostly concentrated on one concept. Experts, unlike intermediates, were also willing to recognize and reconsider inappropriate design decisions. The experts were more efficient and the program written had a better design. Experts had more absorption activities at higher Bloom levels such as analysis and synthesis than intermediates did, while the latter had most absorption activities occurring at the lower Bloom levels such as comprehension and application. Experts used more denial, reorganization and

expulsion activities than intermediate programmers. Experts took part of knowledge as granted, knew better how to analyze the knowledge, and how to apply strategies to use the knowledge, therefore, they spent more time analyzing the knowledge and generating test cases, as compared to intermediate programmers who spent more time learning the knowledge instead.

CHAPTER 6

CASE STUDY ON PROGRAM DEBUGGING

6.1 Introduction

A bug is an error in a program that causes the program's behavior to be inconsistent with the programmer's or the user's expectations. It manifests itself during compile-time or run-time.

Debugging is the process of locating and correcting bugs. These errors can be made in various stages of software development or evolution, such as specification, implementation, or debugging of earlier errors [84].

Typical activities in the debugging process include program diagnosis, error detection, error removal, and testing [179]. Program diagnosis is analogous to a diagnosis by a doctor. The program displays some "symptoms" and the programmer needs to find the "disease" which causes the symptoms. The programmer runs and tests the program to obtain appropriate clues, analyzes the symptoms, and uses diagnostic knowledge to generate reasonable hypotheses. The hypotheses are then tested in experiments. The experiments can either support or contradict the hypotheses. The process is repeated until the programmer narrows down the bug location by using the isolation strategy (see Figure 6.1). After "treating" the disease (correcting the bug), and testing the program, the programmer makes sure that the program is now correct and free from the bug. If not, the process will be repeated until the bug is completely corrected.

In order to make a correct bug diagnosis, programmer must be able to:

- Obtain and understand the meaning of the symptoms of the bugging program
- Identify the appropriate code segments or components involved in the bug
- Identify the process in which the bug is occurring
- Know the most likely cause of the bug
- Evaluate the evidence and select the strategy of debugging

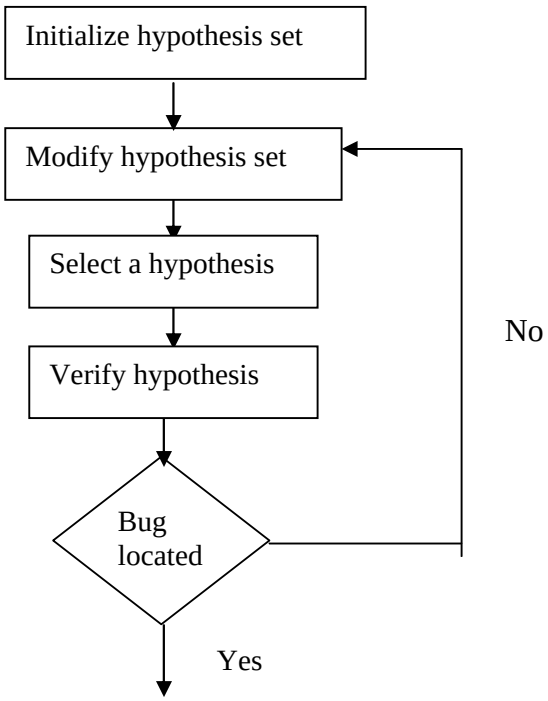

Figure 6.1: A debugging process model (modified from [3])

Program debugging is a common part of software processes. It is an accepted fact that software engineering processes require a significant amount of time for debugging and infected program. In turn, the debugging contains program

comprehension and testing as the central tasks [22] [137]. Therefore, program debugging is a cognitive-oriented process, and studying the cognitive activities involved becomes very important.

6.2 Case Study Design

In order to understand the cognitive activities involved in debugging process, we conducted a case study [178]. The case study design is based on the following hypothesis – "The new learning model based on constructivist learning and Bloom's Taxonomy is applicable for studying the cognitive activities in program debugging", as verified by the case study [172] [173].

In order to validate the hypothesis, we chose one previous case study of a system failure in a COTS (Commercial-Off-the-Shelf) Based Information System[75]. The additional details are in [75].

COTS systems consist of individual components that may be developed by different companies. The probability of failure increases due to integration of individual components. Therefore, debugging a COTS system requires not only the diagnostic skills, but also the knowledge of underlying component technologies.

The case study deals with a system that consists of a COTS Web server and a relational database management system, each provided by a separate vendor. The application was expected to be used by multiple customers, but a serious problem was reported when it was deployed for the first time. The symptoms of the

problem were that the system was either too slow or was not responding at all to user requests. The purpose of the case study was to find and fix the bug.

The programmer was provided with the symptoms “system not responding or too slow”. Based on his experience with the architecture of similar systems, the programmer considered several possible causes. He decided that the most likely cause is the slow database query that slows down access to the web server. The slow web server then blocks access to the site.

Based on this assumption, the programmer used a “browser skeleton” to validate this hypothesis. The browser skeleton is a test harness in which a program simulates customers connecting to the web server. The average response time was recorded after a browser skeleton was launched. Then the actual web browser was launched to connect to the server and the response time of the server was recorded. This turned out to be 100 times longer than the response time for the browser skeleton. This test was repeated several times to make sure that the results were stable and repeatable. The test results supported the hypothesis that queries to the database slow down access to the web server. The programmer had narrowed down the cause of the problem, but further work is needed.

The next hypothesis was that the web server could not spawn child processes to handle customer requests. Based on this assumption, another test was conducted using the same test harness to observe the state of all processes. If the hypothesis were true, the number of processes would not increase after a certain period of time. However, the result did show that some new processes were spawned, and the overall number of processes increased. Therefore, the second

hypothesis was not supported by the evidence. Nonetheless, a useful observation was obtained: One of the child processes was accumulating CPU time and others were not. A further investigation was conducted to see what each child was doing during the browser request process. It was found that only one child works properly and other children tried to set a lock on a file that was already locked by another process. Therefore, there was a block or race condition that put other child processes into an infinite loop and blocked their execution.

After reading the Unix manual and going through documentation and configuration settings, the programmer noticed that the web server uses system calls getcontext and setcontext, which are used in user-level threaded applications. The programmer knew that the web server was actually a multi-threaded application with user-level threads and found that the child process was not threading as expected. Therefore a further hypothesis was made that the actual COTS server agent was not multi-threaded and forced the web server to become blocked.

The server agent provides communication between the web server and the relational database management system. The present server agent provided by the database vendor is used with the www server-side API, although it also supports the Common Gateway Interface (CGI). If the above hypothesis is true, as CGI supports multiple threading, then using the CGI version of the same agent instead of www server-side API version of the server agent would be the right solution. A new test was proposed after the www server-side API version of the server agent had been replaced with CGI version. The test harness and actual web browser were used again. The result demonstrated that the web server was no longer blocked after the

replacement. The problem was diagnosed, a replacement was done and a re-test was conducted. Therefore, the debugging was successful and the bug was fixed.

We separated and classified the debugging process into twenty-eight episodes according to such a rule that, if programmers referred different concepts, a new episode was assigned (see Appendix C). The detailed description of each episode is shown in Appendix C. Then we used self-directed learning theory as the code scheme to analyze the data as described in Chapter 4.

6.3 Case Study Result

Table 6.1 lists the distribution of the cognitive activities and Bloom' levels in this case study. Based on the categorization, the programmer's actions reflect all levels of Bloom's hierarchy, starting with recognition (or knowledge) and comprehension, through application and analysis, and finally synthesis and evaluation.

In terms of cognitive activities, the most frequent activity is absorption which takes more than 67% of total activities during program debugging, since programmers try to understand program in order to perform debugging. Absorption is distributed at various Bloom levels, but there are more cases at comprehension level which takes 25% of total activities. There are also some representatives for reorganization distributed on comprehension, application, synthesis and evaluation Bloom's levels. No denial was found in this case study. Expulsion only occurs at the recognition (or knowledge) level.

Activities at the comprehension level occurred with the most frequency, taking up to 28.56% of total activities. The distributions of activities in application, analysis, synthesis and evaluation levels are similar.

Knowledge level cognition is the prerequisite for program debugging. A programmer must be able to recognize the multi-tier architecture of web application, in order to identify the meaning of the output, and to define the test harness. If the programmer does not have this knowledge, it is not possible to perform debugging.

Armed with this knowledge, the programmer can move to the next taxonomy level – comprehension. The programmer should be able to comprehend how the client-server works, to explain how SQL queries work, to illustrate how the test harness replaces the web server, and to interpret the race condition for two or more threads.

The next taxonomy level is application. By applying the knowledge of three-tier architecture, the programmer attempts to solve the problem by assuming that it is the slow response to SQL query. The programmer also applies the knowledge of how spawning child processes handle each web request.

After the knowledge has been applied, the programmer analyzes the relationships of different pieces of information. For example, after obtaining the first test result and observing the amount of time used for each response, the programmer contrasts the response time with the actual web browser and the response time with the browser skeleton, and finds that the first is 100 times longer.

During debugging, there are several examples of synthesis. The programmer proposes that the hypothesis “slow database query slows down the access to the

web server", then "the web server can not spawn child process", then later "not all child processes spawned were handling inbound requests" and finally "the COTS server agent was not multi-threaded and forced the web server to block". All these hypotheses are based on a synthesis of all the available information.

Table 6.1: The distributions of the cognitive activities and Bloom's levels in the case study

	Assimilation		Accommodation		Total
	Absorption	**Denial**	**Reorganization**	**Expulsion**	
Knowledge	1 3.6%	0 0%	0 0%	1 3.6%	2 7.1%
Comprehension	7 25%	0 0%	1 3.6%	0 0%	8 28.6%
Application	4 14.3%	0 0%	1 3.6%	0 0%	5 17.9%
Analysis	4 14.3%	0 0%	0 0%	0 0%	4 14.3%
Synthesis	1 3.6%	0 0%	4 14.3%	0 0%	5 17.9%
Evaluation	2 7.1%	0 0%	2 7.1%	0 0%	4 14.2%
Total	19 67.8%	0 0%	8 28.6%	1 3.6%	28 100%

On the evaluation level, the programmer validates whether the hypothesis can correctly explain the bug. For example, the programmer validates the hypothesis "the COTS server agent was not multi-threaded and forced the web server to block". In a similar vein, the programmer rejects the second hypothesis "web server could not spawn child processes to handle customer request".

In order to show it more clearly, we extract some examples for each Bloom's level and list them in Table 6.2. In program debugging, knowledge and

comprehension are prerequisites, as the programmer has obtained them prior to debugging the program. The programmer was familiar with programming languages, technologies, and client-server architecture, and was able to obtain the information such as program output. The programmer moved from one level to another in increasing order of complexity. Without establishing a firm footing, it would be difficult to proceed to the next level of cognition.

Table 6.2: Examples at each cognitive level in the case study

Level	Case study behaviors
Knowledge	Structure of the program, recognition of bug symptoms
Comprehension	Understanding how client-server and SQL queries work
Application	Slow response to SQL query causes slow response of the system and suggests that no child processes are created
Analysis	One child process blocks other child processes or the web server does not create multiple processes, which forces no system response
Synthesis	Slow response from web server means slow access to SQL query, and one child process does not handle multiple request
Evaluation	Checking if the original diagnosis explains the entire discovery, Any other possibilities? After replacing the server agent Retest the system with test harness and actual browser

Program debugging might be a complicated process, one which requires the programmer to use all six Bloom's levels. Experts have already had the knowledge and the ability for comprehension; therefore, their efforts mainly concentrate on application, analysis, synthesis and evaluation. We speculate that this is the cause of the large gap between the novices and experts, and this coincides with the work of von Mayrhauser and Vans (1997) [159], who found that programmers who have

more domain knowledge would perform program comprehension at a higher-level abstraction. Taking examples from our case study, we can find that novices might have trouble identifying the cause of the slow response to SQL query, whereas experts immediately recognize it as the problem in creating child process. Once they have accumulated enough knowledge, novice programmers will move more easily to higher levels such as analysis and synthesis.

6.4 Threats to the Case Study

This case study has the following limitations. The case study was based on a previous case study. The description of the case study might have been modified for publication. If the data was obtained from think-aloud protocol, the results might be slightly different. The problem solved in the case study is a small web application. A more complex problem or problems in different domains may produce different results.

6.5 Summary of the Case Study

We used our self-directed learning model to conduct a case study on the program debugging where a large amount of time was spent on program comprehension. The case study result showed that during program debugging, absorption is the driving activity that occurs most frequently and at all six learning levels. Reorganization also appears. Expulsion only happens at recognition

(knowledge) level. No denial activity is found in this case study. All six Bloom's levels have been identified in the case study. Activities at comprehension level take the most with 28.56% of total activities. The distribution of activities in application, analysis, synthesis and evaluation is similar. During program debugging, programmers apply the four cognitive activities at six Bloom' learning levels in order to understand the program and to locate the bug and to fix it. Therefore, the case study has validated the hypotheses stated in the case study design.

CHAPTER 7

CASE STUDY ON SOFTWARE EVOLUTION

7.1 Introduction

7.1.1 Software Evolution

Software evolution refers to the maintenance tasks, one of the important stages of software lifecycle [116]. During software evolution, programmers add new functionality into the existing system of iterated tasks. This process is called the incremental change. Each increment change is based on a change request[118].

Lehman (1998) [88] emphasized that the software evolution becomes very difficult or impossible when software increases its size and that clear steps should be taken in order to keep system stable. Rajlich (2000, 2004) [116] [118] identified three steps for achieving incremental change: concept location, impact analysis and change propagation. Concept location is the first part of software maintenance task and it reconstructs mapping between the application domain and implementation [28]. The purpose is to find the relevant concepts in the code. Impact analysis is to determine the sets of classes that incremental change may affect. Change propagation refers to the process of a programmer changing a class and then visiting all classes in interaction with the changed class in order to investigate whether or not they also need to be changed [116].

While previous papers reported experience with waterfall development, software evolution and maintenance were listed among the topics of the body of software engineering knowledge (SEEK) proposed by IEEE-CS and ACM [80]. This is the basis for the education of software engineers. Since software maintenance takes a substantial part of the software costs and most software engineers are actually maintaining the existing system, it is beneficial for students to work on evolution and maintenance projects. Pair programming described in the section 7.2 exclusively deals with software evolution or maintenance.

7.1.2 Project in Software Engineering Classes

Software engineering is an important part of computer science curriculum at both undergraduate and graduate levels. Undergraduate classes often deal with the fundamentals of software development which include such tasks as the waterfall development model, whilst graduate level offers advanced topics including software evolution and maintenance.

Software engineering in industry is a process of collaboration because most software is developed by a team. That is the reason why there is a team project component in the software engineering classes. According to [20], projects can allow student to learn to develop complex systems, to experience simulated real-world development environment, to get a chance to learn from

each other, to share their skills and knowledge, and to learn how to handle scheduling and other problems.

The traditional way of organizing the project in an undergraduate software engineering class is to form groups with 4-6 students and to use the waterfall software development life cycle to develop a software prototype [68], as does the graduate software engineering class, which requires normally a team work in a project.

Sommervill (1996) [143] pointed out the critical goals of software engineering courses: to make students practice “programming in the large”, to let them practice concepts learned in classroom, and to learn how to use software tools. Peslak et al. (2004) [113] proposed to use collaborative methods to simulate the industrial environment.

According to Gnatz (2002) [68], the course projects using a group larger than 4 people often fail as a result of issues such as scheduling, and personal conflicts. The failure of projects mainly comes from inefficient exchange of information since too many people in a group sometimes cause substantial communication overhead. In fact, the projects are often completed by a few advanced students in the group, and the rest of the team learns less [68].

7.1.3 Pair Programming in Classroom Projects

As mentioned earlier, Pair Programming (PP), one of the important practices of eXtreme Programming (XP), is a collaborative programming [10].

Pair programming has been used in industry and promising results have been reported. Nosek (1998) [105] studied 15 professional programmers in both pair programming and individual programming situations and found that all pairs outperformed the individuals in terms of quality and time spent. William (2001)[169] did a survey of professional programmers and found that 100% were more confident in their solutions when using pair programming. However, recent studies demonstrate that pair programming should not be used for either very large systems [150] or trivial problems [10] [101].

Several researchers used pair programming in undergraduate computer science courses and found it to be beneficial to students. William et al. (2000)[170] performed a case study in an introductory computer science course with 41 undergraduate students who formed two groups: one that worked in pairs and the other one that worked individually. The students were asked to complete four programming assignments. They found that paired students completed assignments faster and with higher quality. The defects of their programs were 15% lower and they also appeared to learn faster than the students who worked on their own [170].

Hedin et al. (2003) [73] used pair programming to teach a second year software engineering class that had 107 students and reported a very positive result McDowell et al. (2002) [96] found that students who worked in pairs implemented better programs and performed significantly better in exams than those worked individually. They also noticed that fewer students in pair programming dropped out of the course than those working by themselves.

Williams et al. (2003) [171] conducted a mass case study in introductory programming courses that involved over 1200 students. It was found that paired students were more likely to complete the introductory course with a grade C or better, and their exam and project scores were better than those who worked individually.

The learning process involved in pair programming has been recently closely studied. Williams and Upchurch (2001) [169] conducted a case study in order to understand how teaching and learning are enhanced by pair programming. They found that the students communicated with each other more effectively, appeared to learn faster, and were happier. The positive effect of pair programming on knowledge sharing during program design was also noticed by others [27].

The benefits of pair rotation have also been observed [146] [27]. The researchers found that students can learn from several partners, while the teachers have to deal with dysfunctional pairs.

However some disadvantages were also reported. Nawrocki and Wokciechowski (2001) [102] claimed that pairs spend nearly twice as much total time than individual programmers. The case study conducted by Gallis et al. (2002) [61] has shown that pair programming is inefficient for trivial programming tasks. The undergraduate students only have basic programming knowledge and they can not conduct pair programming on complex projects.

Schneider and Johnston (2003) [131] noticed that due to the differences of experience between the two programmers in a pair, the stronger one did most

of the job and the weaker one often failed to learn, something that often happens in undergraduate classes. Schneider and Johnston also observed that in order to fully understand the important elements in software development as well as the consequences of agile practices, students must reach certain level of maturity. Otherwise, agile practices such as pair programming might be misused [61] [131].

Pair programming was also used in graduate classes, although not as frequently. Muller and Tichy (2001) [101] performed a case study using pair programming with graduate students who were required to take a practical training course. Twelve participants completed three simple tasks and one project in pairs. They found it easy to adapt to the pair programming in graduate classes. However, some students commented that it was a waste of time to watch their partners dealing with trivial programming problems.

Stotts et al. (2003) [148] did a case study with graduate students and compared the effects of pair programming in a distributed environment and found such an approach both feasible and effective. However, pair programming has rarely been used in graduate software engineering course projects.

7.2 Case Study Design

In order to investigate the effects of pair programming in a graduate software engineering course and in software evolution, and the difference between pair programming and traditional individual programming in that, we

designed a case study where the participants worked either in pairs or individually and made incremental changes to an existing large open source program. That allowed us to make a comparison between the pair programming and traditional individual programming. We also conducted a survey on pair programming [175].

In our design, we were led by our belief that the graduate students have sufficient required knowledge of programming for pair programming, and also have an experience in cooperation which lessens the possibility of failure. We wanted to see how well the graduate students could actually perform in graduate software engineering course projects when using pair programming technique and how effective it would be to integrate the software evolution in open source software project.

The experiment applies a single-factor design [29]. The controlled independent variable is whether the experimental subjects use pair programming or use traditional individual programming [62]. Each subject of both groups solves the same problems and works under the same or similar conditions. The dependent variables for each subject are 1) the total time taken for each change request 2) the lines of code for each change request written by programmers 3) the number of classes they made change necessary 4) the number of classes they made change unnecessary.

The case study design is based on the null hypotheses later falsified by the case study: “Pair programming is not useful for graduate software engineering class” and “Pair programming cannot be conducted with software

evolution projects". The rest of this section describes various facets of the case study design.

7.2.1 Participants

Six graduate students from the Wayne Sate University Department of Computer Science participated in the study. It was conducted through their course project. The case study lasted for one entire semester. The students were experienced in programming in C, Java, or C++, but they had never before performed pair programming, nor did they have experience in software evolution. The programming experience of all the six students is shown in Table 7.1. In the pair group, there are 3 Master students and one Ph.D. student. Two male students and two female students formed two pairs respectively. Since all of them had similar experience in OO programming and had never used pair programming technique before, there is no obvious bias in such pairing. Two master students were in the individual group, one male and one female.

Table 7.1: Programmers' experience

	Pair 1 (A)	**Pair 1 (B)**	**Pair 2 (A)**	**Pair 2 (B)**	**Single 1**	**Single 2**
Years using Java language	3	1	2	5	3	2
Years using C++ language	5	4	1	2	2	2
Other languages known	Matlab	C, VB	C	Perl, Clips	C	C
Total lines of code having written in java	>1000	500	<1000	2000 - 3000	2000	>1000
lines of biggest programs having previously studied	>3,000	>1,000	5,000	>1,000	1000	1500

7.2.2 Material

The material for this case study is an open source project called JAdvisor [144]. JAdvisor is a class scheduler, course planner, and course search program written in Java. JAdvisor allows college students to view their schedules graphically and to create an optimal schedule. We selected this open-source application since it has reasonably big size with 31 classes, and satisfies Beck's criterion that the tasks for pair programming should take at least several hours [10]. JAdvisor application was also new to all the programmers.

The students were working on six distinct change requests that varied in difficulty. Here are the change requests:

Change request 1: Color coding schedule. JAdvisor allows the user to input block time to the schedule; however, there is no specific drop down menu to do so and each classification shows up with same color and is therefore, hard to recognize. We are to add a classification drop down box to the block time menu. It should display various classifications like study time, food break time, homework time, etc., with different color blocks on the schedule. Via this way the user can easily identify their allocated times.

Change request 2: XML output. Current version of JAdvisor does not support XML output for class scheduling, but we need it to output the schedule of the user. The XML structure must be clear and readable. Once this structure is documented the user should be able to save their output with the XML format,

and also the program should be able to read the XML format files into both the planner and schedule tabs.

Change request 3: Planner wizard. JAdvisor has no planner wizard for the required courses needed for a degree. We are going to create a planner wizard that allows the user to enter all courses necessary for his or her degree. The information should be saved in XML format whenever the user chooses to do so. These courses should be shown in the planner tab on the right hand side. If a course has been taken, the course name should appear in red color and the user should not be allowed to select it.

Change request 4: Planner duplication. The planner is supposed to allow the user to plan all four years of the curriculum, but it allows for duplicates. Since students usually do not repeat their courses, JAdvisor should ask the user to overwrite if he or she does add a duplicate course either in the same semester or in future semesters.

Change request 5: Pop-up dialog. The planner does not allow the user to select a course and then to display its information. We are going to add a pop-up dialog window which allows the user to click on a course in the current term and then a pop-up dialog should be able to display the course information.

Change request 6: HTML input. The current version of JAdvisor allows the user to output HTML, but it cannot read HTML files. The program should be able to read HTML files and fill in the scheduler as if they were saved.

7.2.3 Procedures

Prior to the case study, all subjects were given lectures on basic concepts of pair programming and incremental change such as concept location, change propagation and impact analysis. They were taught how to do incremental change and how to conduct pair programming. They were also provided with JAdvisor website and the change requests.

The author acted as a mentor who monitored the programming process for all the pairs. The programmers were required to document the time they used for their tasks.

One pair and one individual programmer worked on change requests 1, 2 and 3, and the other pair and the other individual programmer worked on change requests 4, 5 and 6. They worked on those change requests in sequence.

After the case study was completed, programmer pairs and individual programmers were asked to write a report on the process with screen shots. Based on the observation, the program written by programmers, and the reports, we conduct analysis.

7.3 Case Study Result

In this section, we summarize the results of experiments and student survey on pair programming.

7.3.1 Case Study Results

All six participants have completed their work. In Table 7.2, we presented the time the participants spent on each change request. Please note that the times listed in Table 7.2 are the duration that the pairs and individuals spent on the tasks. Therefore, the "total time cost" for a pair is double the time indicated. In all cases, it is bigger than the total time cost spent by individuals who worked on the same tasks. On the other hand, programmer pairs worked on the tasks for a significantly shorter duration than individual programmers. This indicates the pairing can reduce the time of software evolution, which is consistent with the result for pair programming in software development reported by [34, 170]. However it may increase the cost of the development as the "total time investment" for two programmers is higher.

Table 7.2: Comparison of time (hours) used by pair and individual programmers

Subjects	**Tasks completed in order**	**Pairs**	**Individuals**
Pair 1/ Individual 1	Change 1	12	15
	Change 2	9	12
	Change 3	9	15
Pair 2/ individual2	Change 4	13	15
	Change 5	11	15
	Change 6	7	14

Change requests 1, 2 and 3 were done by one pair and change requests 4, 5 and 6 were done by the second pair in sequence. From Table 7.2, we can see that the time spent by the pairs on the change requests became shorter as

the process moved along; however, this phenomenon is not shown by the work of the individuals. This indicates that the learning process is faster for the pairs than for the individuals. This result agrees with the results claimed by Williams et al. (2000) [170], who stated that pairing forces the two programmers to think aloud, making their comprehension strategy explicit, and thus, enhanced the learning process.

Table 7.3: Change request results for pairs

Change requests	**Pairs**		
	Lines of code	**Numbers of classes changed**	**Numbers of classes changed unnecessary**
1	110	3	1
2	87	3	0
3	56	1	0
4	64	4	0
5	82	3	0
6	67	3	0
Total	466	17	1

Table 7.4: Change request results for individuals

Change requests	**Individuals**		
	Lines of code	**Numbers of classes changed necessary**	**Numbers of classes changed unnecessary**
1	125	3	0
2	104	3	0
3	86	1	2
4	102	3	1
5	78	3	0
6	75	2	0
Total	570	15	3

Tables 7.3 and 7.4 show the numbers of lines of code added for each change request, the numbers of classes which have been correctly changed,

and the numbers of classes which should not be changed but were modified by the programmer pairs or individuals. For six change requests, pair programmers added a total of 466 lines of code, but there are 570 lines of code written by individual programmers. The pair programmers also did most of the job correctly since they completed more required change propagations than individual programmers. The individual programmers failed to find 3 change propagations, and did unnecessary change in 3 classes. But the pairs only failed to find one change propagation out of total eighteen. Also the programs written by pairs have more meaningful variable names. This proves that with the help from the partner, programmer pairs are able to write higher quality code, supporting the same finding by [170].

Table 7.5: Result of quantitative questions from the survey for pairs

Questions	**Lowest (0.0)**	**Highest (5.0)**	**Answer (average)**
How effective do you think pair programming was for the project?	Not effective	Very effective	4.00
Did you and your partner contribute equally to the project?	Very unequal	Equal	3.20
What is your rating of your performance?	Did very little	Did most of work	3.8
What is your rating of your partner's performance?	Did very little	Did most of work	3..9
Do you think you learned more or less than you would have if you had worked on your own?	Much less	Much more	3..8
How do you think the time that you personally spent on this project compares to the time it would have taken you to do it on your own?	Pairs much slower	Pairs much faster	4.0
Would you like to use pair programming during your future graduate course project?	Not at all	Very like	4.6

7.3.2 Survey Results

A survey on pair programming was conducted with the survey questions listed in Table 7.5 through Table 7.7.

In general, students thought that pair programming is an effective way to complete the course project. All students were satisfied with their own performance as well as that of their partners; however, they thought the contributions from the two people in the pair were different, which is consistent with the two roles in the pair: One is the driver who controls the keyboard and the other is an observer performing code review in helping the driver to make decisions [10]. It seems that a role rotation is needed in order to further promote learning between the two programmers in the pair as mentioned in [146]. The students also believed that pair programming can actually save their time in completing the projects. This corresponds to the shorter project durations that were indicated in Table 7.2. The two pairs reported that they enjoyed the pair programming technique more than them programming alone and are willing to apply it in their future projects. It seems that the programmer pairs had higher motivation to finish the tasks than did individual programmers, also contributing to the time difference on the tasks by the pairs.

The two individuals who participated in the case study were also conducted a survey via email. The questions and results are shown in Table 7.7. It seems that the two individuals expected positive effects in applying pair

programming in the course projects and they were also interested in using it in the near future.

Table 7.6: Answers of qualitative questions from the survey for pairs

Questions	Answers
Do you feel like you have learned anything just by reading your partner's code?	• Yes, I think so • Yes, I agree • No, I do not feel that way • No, I did not learn more
What was the biggest problem you have had to overcome as a paired programmer?	• No problem • I do not think there is a problem • The pair should know how to deal with different options • Partners should have similar knowledge
What do you think is the biggest concern in pair programming?	• Matching of the two programmers • Having agreement • Not being ready for big problem • Time conflict
What are the advantages of pair programming?	• Pair works faster • Pair explores more design and implementation options • Pair has more confidence • Pair can share ideas

Based on the case study and survey results, it seems that pair programming and the large application like JAdvisor can work as graduate software engineering projects since the paired programmers produced higher quality results within shorter time, compared with those working alone. The programming ability of the pairs was better than that of individuals based on Tables 7.2, 7.3, and 7.4.

Using pair programming, learning has been enhanced greatly, since the knowledge is constantly exchanged between partners, including those concepts learned in the classroom, tool usage tips, programming language, design skill,

and debugging techniques. Pair programming affects learning and motivation, facts that have been demonstrated by the result of the survey.

Table 7.7: Answers of qualitative questions from the survey from individuals

Questions	Answers
Do you think it would be more efficient if you would have worked with a partner?	• Yes, I think it would • Yes, I agree, but not sure
Do you think you learn more if you would have worked with a partner than when you worked alone?	• Yes • Yes
How do you think the time that you would spend with a partner compared to the time you spent on your own?	• It depends on the partner. If the team was on the same level, it would be more efficient and require lesser time. But if the partner was learning then I don't see any advantage. • Possibly we would use less time by working as pair
How do you think the pair programming impacts a quality of the program?	• Better than with a single person project • It should have higher quality
If you are allowed to choose in future course projects, would you prefer to work with a partner or work alone?	• With a partner • I love to pair with a friend to see the results

Nevertheless, using the pair programming concept in graduate class projects certainly has some obstacles. The first one is how to pair students. Letting students themselves decide who to pair with is often a good option that can minimize incompatibility. The second issue is the types of programming problems for the projects. Theoretically, any program problem with medium size is suitable for pair programming. However, some problems may motivate students to learn more than others. The third obstacle is the scheduling problem. If the two students have scheduling problems, they cannot work on the project together.

7.4 Threats to the Case Study

Several issues affected our results and limit our observations of the case study. These stem from the nature of any case study, and from our general approach.

Although the Java program JAdvisor is an open source project, it is not representative of programs as a whole. It is not a very large program and in no way is able to represent all OO programs. Much larger size of programs might have slightly different results.

The change requests given were open to the interpretation of the participants. While all questions pertaining to the requirements were answered, several students felt that some of them were not required or were loosely coupled to the more important change. One example is the implementation of the link between the scheduler and planner components without a drop down box to determine which semester to add the course to. This could also be the result of a poor assumption. The JAdvisor program on its first execution asks for the current semester, which validates the assumption. However, in future executions, the user can load saved data, which will change the current semester. Therefore the assumption does not hold for consecutive executions of the system.

7.5 Summary of the Case Study

Pair programming increasingly attracts attention not only in the industry, but also of academia. Pair programming has been reported to have benefits in software development and in undergraduate classroom.

Because of their maturity, a good command of programming knowledge, experience in cooperation, and the moderate size of graduate software engineering class projects, graduate students are the ideal candidates to use pair programming in their classroom projects as well.

We performed a case study on pair programming in software evolution at graduate software engineering class with six graduate students, who worked as either pairs or individuals. All the subjects worked on change requests on open source software: JAdvisor.

The case study showed that programmer pairs consume shorter time, but at an increased cost because of the extra labor involved. Programmer pairs completed the tasks with higher quality work than those done by individuals since they only missed one change propagation out of eighteen in total, but individual programmers missed three. They have also written higher quality code, such as with less lines of code and more meaningful variable names. According to the survey, all students are satisfied with the performance by themselves and their partners and they are willing to use pair programming in the future. Based on the case study and survey results, it seems that it is a good idea to develop graduate software engineering course project with pair programming and pair programming can be applied to software evolution tasks

which falsifies the hypothesis. Using pair programming technique to organize the projects not only forces students to learn, but also gives them some fun.

CHAPTER 8

EVALUATION OF DIALOG-BASED PROTOCOL, SOFTWARE SCREEN-CAPTURING AND SELF-DIRECTED LEARNING THEORY

Since we used a novel empirical technique of dialog based protocol, software screen capturing and self-direct learning theory to conduct our empirical study, in this section we summarize our experience with that technique and evaluate our novel empirical approach.

8.1 Dialog-Based Protocol

According to our observation, when two programmers in a pair were well matched, they worked efficiently and conducted a revealing dialog. However in our case study on incremental software development, we ran into some difficulties which might indicate the issues and limitations of the dialog-based protocol.

One pair did not use the dialog to communicate, and only one programmer spoke all the time. That programmer wrote the program alone and explained his thoughts to the other member. His partner simply said “yes” or “right” during the process, and it looked as if this pair had simply applied the think-aloud protocol. We considered these data to be invalid and did not include them in the final analysis of that case study.

One pair did not follow the directions to implement an object-oriented program and only created one class for the whole program. Again we considered the resulting data to be invalid.

One pair had experienced great difficulty working together although they did manage to complete the task. There were a lot of arguments and a non-task-related discussion. Those arguments affected their motivation and impacted the results. We still considered the data to be valid after we removed the irrelevant parts of the dialog.

We often observed cases where one subject did not understand the partner's idea and the partner tried to illustrate and explain it to the partner in more detail. Whenever an ambiguity or misunderstanding arose, the other partner would force the subject to correct it or make it clear. Based on this observation we are convinced that the data collected in dialog-based protocol are more complete and more correct than in think-aloud protocol, which confirmed our earlier anticipation.

Since the mentor was simply an observer, the communication between the mentor and the subjects had been greatly reduced. Even when subjects had a technical question, they did not have to ask the mentor for help since their partner was often able to solve the problem. During the case study, the mentor was only asked questions at the beginning of the process when several pairs had problems with the use of the compiler. There was no communication at all between the mentor and three pairs. Therefore, the placebo effect was greatly reduced.

The presence of the mentor in the dialog-based protocol was not necessary, particularly when subjects fully understand program requirements and are familiar with the tools. We observed that programmer pairs talked quite naturally and frankly and some of them often joked, which indicated that they totally forgot that they were conducting the experiment. Therefore, the experiment was conducted in more natural environment and the Hawthorne effect was minimized.

In general, our observations during the case study directly supported the anticipated comparisons between think-aloud and dialog-based protocols shown in Table 2.1. We concluded that dialog-based protocol is an effective method for collecting information on the programmers' cognitive activities during software engineering processes, although there are still some open issues, such as how to form the pairs.

8.2 Software Screen-Capturing

In our case study on incremental software development and software evolution, we used videotaping for the first two pairs and then switched to software screen-capturing for the rest. Videotaping provided more data, but these additional data did not play any significant role in our case study.

Recording quality with a video camera turned out to be lower than using a software screen-capturing. The camera had to be located a certain distance from the subjects in order to capture the whole computer screen, but that increased

the external noise in the recordings. Some screen data were missing since the camera was located behind the subjects and when subjects moved, their bodies hid parts of the screen. There were no such problems with software screen-capturing.

We also found that screen-capturing reduced the Hawthorne effect. One pair was clearly camera shy since they occasionally asked if the taping has been stopped, indicating a possibility of the Hawthorne effect. We found no evidence of similar awareness during the software screen-capturing.

Transcription of the videotape took much more time than using software screen-capturing. There were several procedures and particular software we had to use in order to transcribe the digital tapes into media files. Playing back the tapes was more time-consuming and complex than playing back media files.

Videotaping needed a detailed time schedule in which two programmers in the pair, the mentor, and the video-camera all had to be available at the same time.

The media files captured by software did not always display continuous scenes on the screen due to the delays of capturing. However, no significant data was lost in this way, and we did not have any difficulties transcribing media files into dialogs.

8.3 Code Scheme: Self-Directed Learning Theory

Through the case study, we found that the self-directed learning model provides a clear and easy-to-use coding scheme in studying the cognitive activities during software engineering processes. It can be used to classify the cognitive activities and learning levels used by programmers during incremental software development and program debugging, and thus to distinguish experts and novices.

Similar to other coding schemes, the self-directed learning theory leaves room for disagreements. A coding scheme uses a limited selection of verbs for the four cognitive activities and six Bloom learning levels (Tables 4.1 and 2.2). The verbs for the Bloom levels have been developed and validated through many years of use, but still some verbs appear at more than one Bloom levels, creating an ambiguous classification process. The verbs for cognitive activities in Table 4.1 were used in our case studied for the first time and might not be complete. About 10-15% of episodes in recorded dialogs did not contain those verbs or even their synonyms. Therefore, not only the verbs in the episode, but also the actual contents of the episode were considered during the classification.

CHAPTER 9

CONCLUSIONS AND FUTURE WORK

Although we have had a conclusion for each chapter, this part will serve as an overall conclusion of the thesis that corresponds to the general objectives. We describe the conclusions under the categories of dialog-based protocol, self-directed learning theory, and cognitive process during incremental software development, cognitive process during program debugging, and pair programming in software evolution project. Each of them will be presented respectively in sections 9.1, 9.2, 9.3, 9.4 and 9.5. After that, in Section 9.6, we will elaborate the future work that can extend the subjects of this thesis.

9.1 Dialog-Based Protocol

Based on the idea of pair programming, we proposed a dialog-based protocol, which is an alternative to common think-aloud protocol. Through the case study, we found that we can collect more complete and accurate data through dialog-based protocol. Dialog-based protocol can also reduce the Hawthorne and placebo effect because it creates a more nature dialog environment and reduces the communication between the mentor and subjects. Therefore, it is considered as an effective method for collecting information on the programmers' cognitive activities during software engineering processes.

9.2 Self-Directed Learning Theory

Based on the constructivist learning theory, we classified the cognitive process into four cognitive activities: absorption, reorganization, denial and expulsion. We also used the Bloom's taxonomy of cognitive domain to classify the learning levels. The result of the case study demonstrates that the self-directed learning model provides a clear and easy-to-use coding scheme in the study of cognitive activities during software engineering processes.

9.3 Cognitive Process during Incremental Software Development

In order to study how a programmer learns and what is the cognitive process during incremental software development, we applied a novel empirical method that consists of dialog-based protocol, software screen-capturing, and a coding scheme based on self-directed learning.

From the case study, we found that all intermediate level programmers followed a similar cognitive process with a similar distribution of the cognitive activities and Bloom levels. Using incremental software development with the test-first approach, absorption is the dominant activity behind the process. Reorganization often occurs, but denial and expulsion occasionally appear. The four cognitive activities appear at four of the six of Bloom's levels.

Compared to intermediate programmers, experts discuss more domain concepts at a greater length of time before they start writing their code. Experts concentrate mostly on one concept at one time, while intermediate programmers

often discuss several concepts simultaneously. Experts are willing to reconsider and correct obsolete design decisions, while intermediate programmers retain all design decisions. Experts have more absorption activities at higher Bloom levels, such as analysis and synthesis, whereas intermediate programmers had more absorption activities at lower Bloom levels, such as comprehension and application. Experts spend more time analyzing the knowledge and generating test cases, intermediate programmers spend more time learning the knowledge.

9.4 Cognitive Process during Program Debugging

In this case study, we studied the cognitive aspect of program debugging using self-directed learning theory. The activities in program debugging were classified according to the four cognitive activities at six levels in Bloom's hierarchy. The case study showed that for debugging, programmers have to utilize all six levels of Bloom's hierarchy in order to be able to make the necessary updates. This indicates the program debugging is a complex and difficult task.

The case study result also indicated that during program debugging, absorption is the driving activity that occurs most frequently and at all six learning levels. Reorganization also appears. Expulsion only happens on knowledge level. No denial activity is found in this case study. All six Bloom's levels are identified in the case study. Activities at comprehension level take up to 28.56%, the biggest portion of total activities.

We speculate that novices may spend more time in accumulating knowledge and in comprehending. In other words, novices are more likely to spend more time on the two lowest levels of the hierarchy. By contrast, experts, who are already confident with this knowledge and in comprehension, concentrate on application, analysis, synthesis and evaluation. We believe that this is what forms the large gap in the performance of novices and experts.

9.5 Pair Programming in Software Evolution Course Projects

Pair programming has been used in undergraduate classes in order to develop student skills and to enhance student learning. Experiments with such an approach have demonstrated positive effects. This case study investigates the effects of pair programming in graduate software engineering classes with six students who were assigned to work on incremental changes on an open source application either as pairs or as individuals. During the experiment, we found that paired students completed their change request tasks faster and with higher quality than individuals. They also wrote less lines of code and used more meaningful variable names. The result of the case study shows that pair programming could be an effective and useful approach for graduate software engineering classes.

Because of maturity, a good command of programming knowledge, experience in cooperation, and the moderate size of graduate software

engineering class projects, graduate students are ideal candidates to use pair programming in their classroom projects.

Graduate software engineering class projects are normally of moderate size. Graduate students are both mature and experienced in cooperation and most of them have a good command of programming knowledge. All the above factors make pair programming ideal for graduate classroom projects.

9.6 Potential Future Work

In the future, we will be using our empirical approach to study other software engineering processes where dialog-based protocol (i. e., pair programming) is applicable. In particular, we want to study more detailed activities that constitute software evolution, such as change request analysis, concept location, impact analysis, change propagation, regression testing, and others. We are hoping to be able to distinguish those different software engineering activities.

Based on the self-directed learning model, we plan to build a documentation tool which can be used to capture the domain knowledge and programming knowledge during incremental software development. The knowledge which is not preserved in the code, but appears in the development process, might be useful for maintenance and debugging as well. Therefore, documenting the knowledge during software development is necessary.

We also plan to conduct research on software evolution in large programs in order to understand the impact of the program size on the cognitive activities of the programmers.

APPENDIX A

PILOT STUDY ON INCREMENTAL SOFTWARE DEVELOPMENT

1. Introduction

At the beginning of this research, we conducted a pilot study on incremental software development [119] [120]. For simplicity, we decided to concentrate on the eXtreme Programming process [10] where pair programming is one of the recommended practices. We use one XP programming example, which was done by two experts who have been working in industry for more than 25 years [93].

Pair programming offers a unique opportunity to study the parallel construction of both the program and the programmer knowledge. In pair programming, the programmers communicate with each other about the evolving program. By recording and analyzing their dialog, we study how both the knowledge and program grow.

The programmers take change requests one at a time and make corresponding program changes. The changes that the programmers perform on the programs fall into three categories: additions of the new functionality called incremental changes, deletions of functionality called retractions, and restructurings of the program called refactorings. Of them, the incremental change is done in direct response to a change request, while retraction and

refactoring are done in response to the whole history of accumulated incremental changes and resulting program structure.

We [119] demonstrated that the knowledge growing process in incremental software development is analogous to constructivist learning (Table A.1). A programmer constructs the program based on the sequence of change requests, such as additions of the new functionality (*incremental changes*), deletions of functionality (*retractions*), and restructuring of the program (*refactoring*). The corresponding cognitive activities are *absorption, expulsion*, and *reorganization*. Sometimes, a programmer might find a change request that is impossible to accomplish, that change request then has to be rejected. The cognitive activity for that is *denial*. We [119] also found that *absorption* is the dominant activity behind the process during incremental software development. *Reorganization* often occurs, but *denial* and *expulsion* only appear occasionally.

Table A.1: Analogy of incremental software development and constructivism

Incremental development	**Constructivist learning**
Programming activity	Cognitive activities
Incremental change	Absorption
Rejection of change request	Denial
Refactoring	Reorganization
Retraction	Expulsion

2. Pilot Study Result

The program developed during this case study records bowling scores [93]. The programming pair started the process with the preliminary knowledge of the domain (bowling rules). This preliminary knowledge is represented in Figure. A.1, where concepts are represented by rectangles and the arrows represent the

dependencies. The dependencies stand for the order in which the concepts have to be explained to one not familiar with the domain. A concept can be explained only if all previous concepts it is dependent on have been explained and understood. Gruber (1993) [71] discussed issues of defining a common theoretical framework and vocabulary so that interested agents can make and share a particular ontological commitment, such as classes, functions and other objects. For simplicity, we use UML notation to represent design decisions in our work.

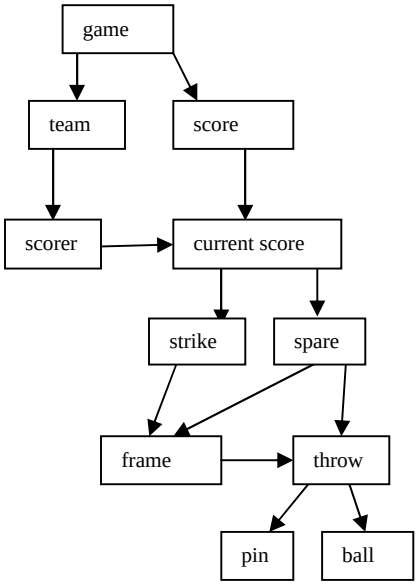

Figure.A.1: Domain concepts in the pilot study

This original domain knowledge serves also as a backlog of things to do, as the programmers intend to implement a program with functionality that fully covers this domain knowledge. Preliminary programming knowledge includes knowledge of the programming language, algorithms, eXtreme Programming practices, and so forth.

Equipped with this knowledge, the programmers implemented a sequence of the program versions and recorded their dialog. The UML class diagram [57] of the first version is in Figure A.2. The design decisions that led to this diagram were extracted from the recorded dialog and appear in Figure A.3. The rectangles represent design decisions while arrows represent the order in which the design decisions were made. Dark rectangles represent domain concepts that serve as the basis of some of the design decisions.

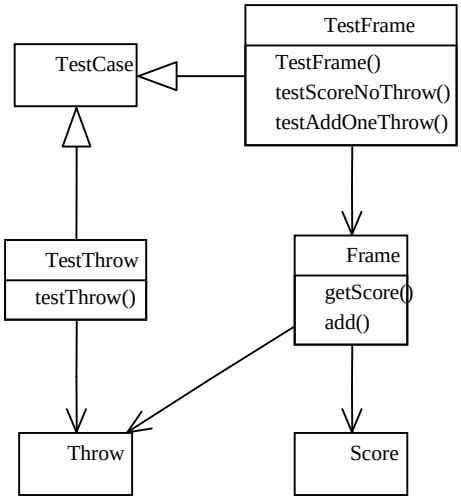

Figure A.2: UML class diagram for the first version

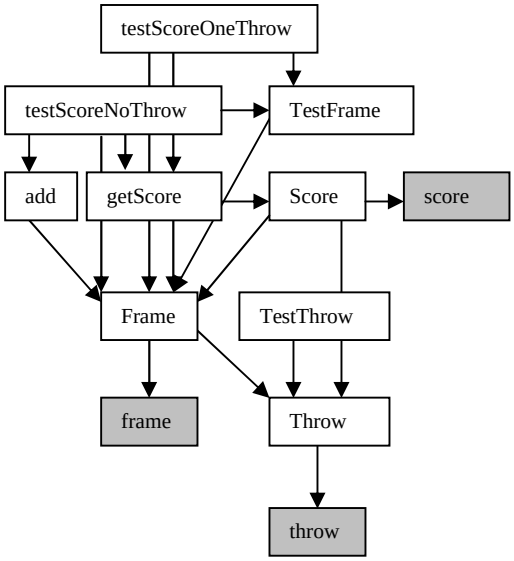

Figure A.3: Design decisions for the first version

Please note that the order of the design decisions does not correspond to the order of the dependencies in the UML diagram. If the two orders were identical, that would mean that all design decisions were done in bottom-up order. The development continued through several steps; see the progress of the learning in Figure A.4, Figure A.5, and Figure A.7. The resulting UML class diagram of the program is in Figure A.6.

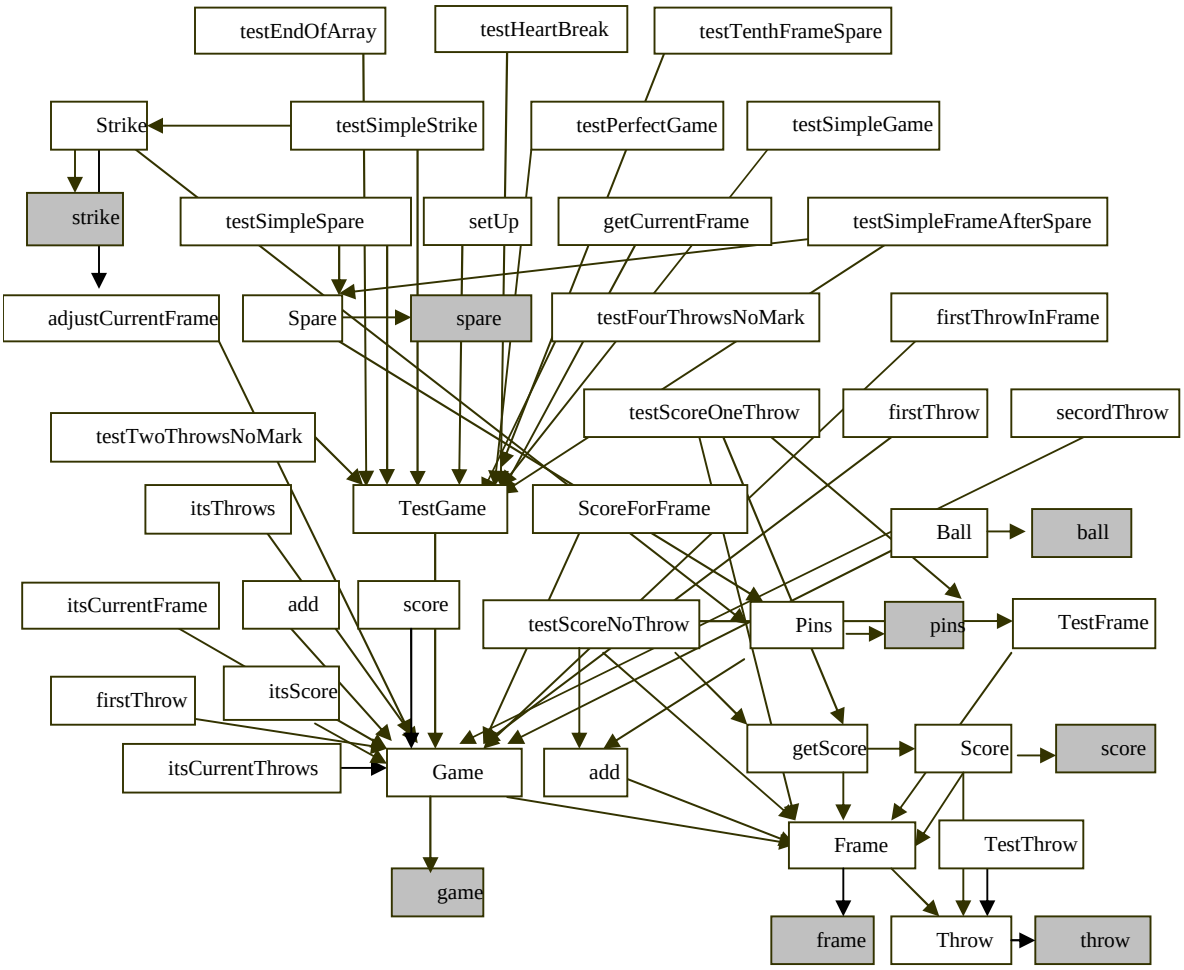

Figure A.4: Design decisions in the middle of development

We divided this dialog into 84 small episodes. Two independent observers classified each episode as one of the four cognitive activities and one of the six levels of Bloom's taxonomy, using the verbs in the dialog as the clue, see Tables 2.2 and 4.1. There were disagreements in approximately 10% of the episodes. There was also an observer bias that led one observer to score higher on Bloom's scale than the other, with the average score being 0.5 higher. While

acknowledging these differences, we found that, in general, the classification was acceptable and reliable.

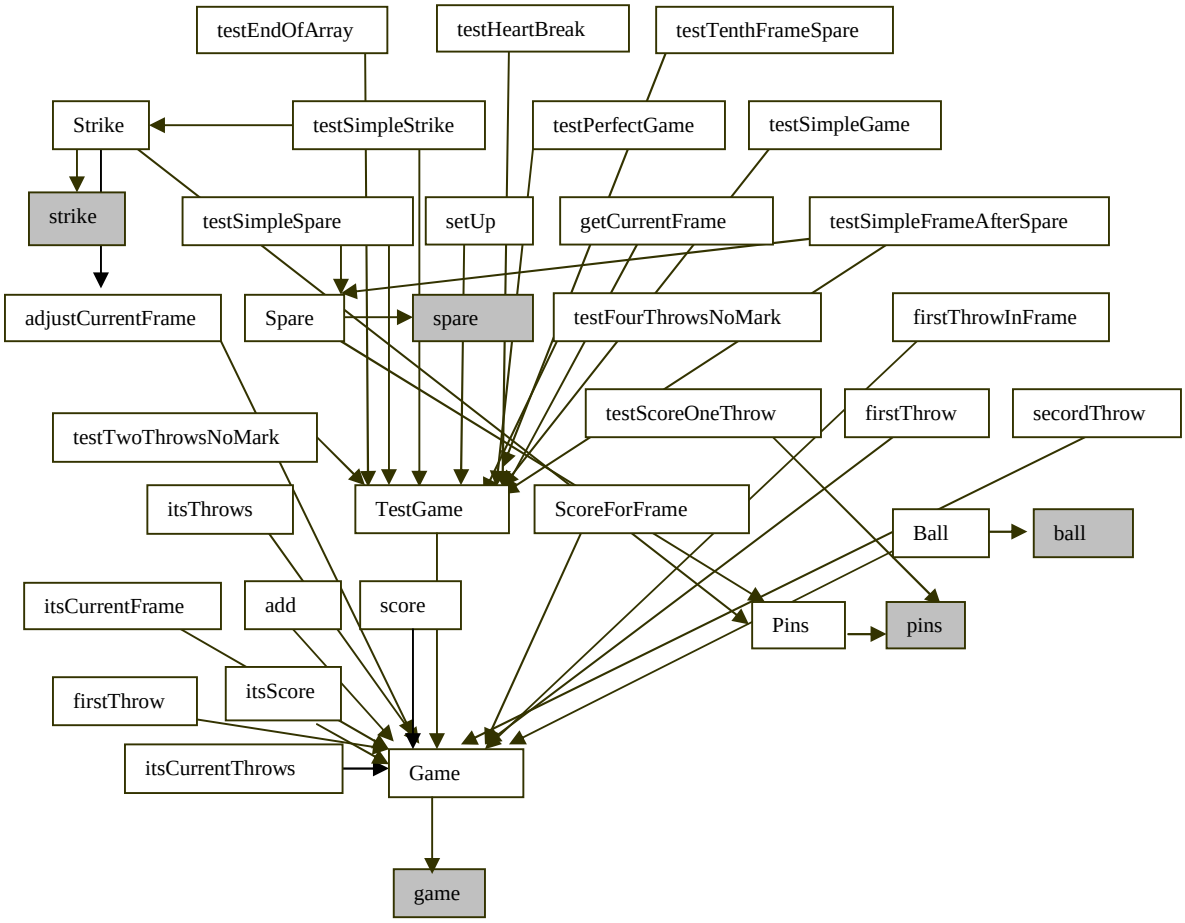

Figure A.5: Design decisions after class Frame was abandoned

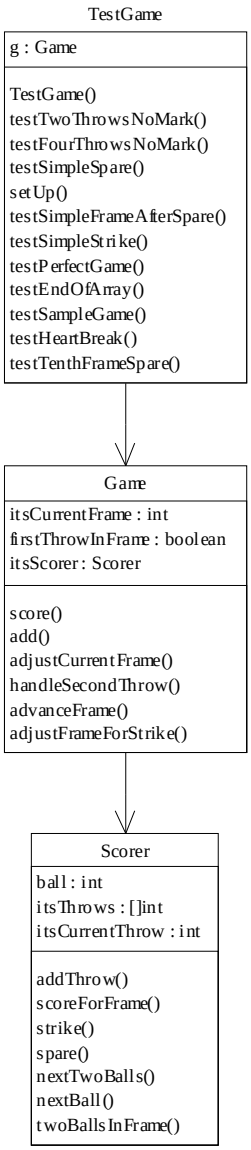

Figure A.6: UML class diagram for the final program

3. Discussion of the Result of the Case Study

In the case study, we observed that the knowledge required by even a small program is quite extensive, as demonstrated in Fig. A.4. It should be remembered that these figures are only the “tip of the iceberg”, as there is a

large preliminary knowledge of the domain and programming that these figures do not capture. Yet all that knowledge is necessary to evolve the program.

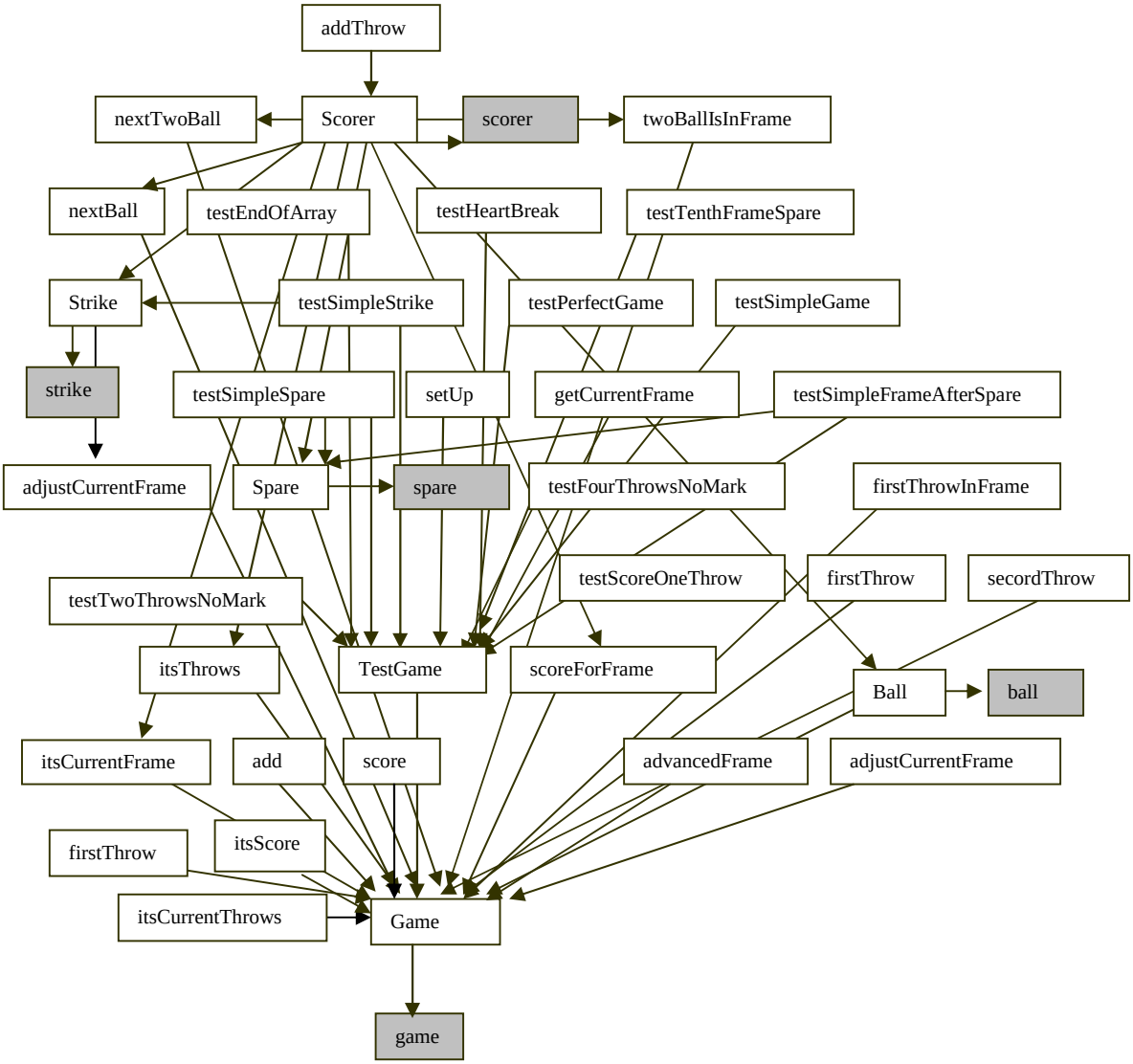

Figure A.7: Design decisions at the end of development

During the incremental program development, the most common cognitive activity was absorption. While absorption was the driving force of the learning, there was one large episode of knowledge expulsion in the last part of the dialog. Class “Frame” of Figure A.4 and six related design decisions out of the total of

39 were retracted, resulting in the knowledge change. This episode illustrates the non-monotonic nature of self-directed learning. The changing number of concepts and their dependencies is presented in Figure. 5.2.

There were also episodes of reorganization of the knowledge. The rejection of concepts Score Card and Team in the first part of the dialog are examples of denial.

We observed that the absorption occurred most often at the beginning and the middle of the dialog, while the expulsion happened most often in the last part. Reorganization appeared mostly in the middle and last parts and denial appeared only in the first and middle parts of the dialog. The episodes belong to all levels of Bloom's hierarchy except the lowest and highest one.

Table A.2. shows that the episodes which were classified according to the self-directed learning theory. Each episode shows either creating new concepts or revisiting and updating existing ones. Almost all the domain concepts were revisited and updated. The domain concept "Frame" was updated as many as 9 times, with each update resulting in more accurate knowledge. Many design decisions were also updated, but less frequently than domain concepts.

As a result of the case study, we conclude that the self-directed learning model explains the learning that takes place in incremental software development.

Table A.2: The classification of the cognitive activities and Bloom levels in the case study on incremental software development

(A: absorption; d: denial; r: reorganization; e: expulsion; 1-6 refers to the six Bloom's levels starting from knowledge)

No	Programmers' action	Bloom's level	activity
1	M: What I'd like to do is write an application that keeps track of a bowling league. It needs to record all the games, determine the ranks of the teams, determine the winners and losers of each weekly match, and accurately score each game. K: You rattled off several user stories, which one would you like to start with.	2	a
2	M: Let's begin with scoring a single game K: What does that mean? What are the inputs and outputs for this story? M: It seems to me that the inputs are simply a sequence of throws. A throw is just an integer that tells how many pins were knocked down by the ball. The output is the data on a standard bowling score card, a set of frames populated with the pins knocked down by each throw, and marks denoting spares and strikes. The most important number in each frame is the current game score	3	a
3	K: Let me sketch out a little picture of this score card to give us a visual reminder of the requirements. K: but it will serve as a decent acceptance test M: We'll need others, but let's deal with that later. How should we start? Shall we come up with a design for the system? K: but I wouldn't mind a UML diagram showing the problem domain concepts that we might see from the score card. That will give us some candidate objects that we can explore further in code."	3	a
4	M: Clearly a game object consists of a sequence of ten frames. Each frame object contains one, two, or three throws. K: Great minds. That was exactly what I was thinking. Let me quickly draw that, but if you tell Kent, I'll deny it.	2	a
5	K: Shall we start at the end of the dependency chain and work backwards? That will make testing easier M: Sure, why not. Let's create a test case for the **Throw class.** K: Do you have a clue what the behavior of a **Throw** object should be? M: It holds the number of pins knocked down by the player	5	a
6	K: Maybe we should come back to it and focus on an object that actually has behavior, instead of one that's just a data store. M: You mean the **Throw** class might not really exist?" K: Well, if it doesn't have any behavior, how important can it be? I don't know if it exists or not yet. I'd just feel more productive if we were working on an object that had more than setters and getters for methods.	4	e

7	M: let's move up the dependency chain to **Frame** and see if there are any test cases we can write that will force us to finish **Throw**	4	a
8	K: Okay, new file, new test case M: Now, can you think of any interesting test cases for **Frame**? A frame might provide its score, the number of pins on each throw, whether there was a strike or a spare... M: the test case passes	5	a
9	M: But **Score** is a really stupid function. It will fail if we add a throw to the frame.	4	a
10	M: So let's write the test case that adds some throws and then checks the score	5	a
11	M: That doesn't compile. There's no **add** method in **Frame.**	2	a
12	K: I'll bet if you define the method it will compile. M: This doesn't compile because we haven't written the **Throw** class	4	a
13	K: The test is passing an integer, and the method expects a **Throw** object.	2	a
14	K: Before we go down the **Throw** path again, can you describe its behavior? M: didn't even notice that I had written **f.add(5)**. I should have written **f.add(new Throw(5))**, but that's ugly as hell. What I *really* want to write is **f.add(5)**. K: Can you describe any behavior of a **Throw** object — binary response. M: I don't know if there is any behavior in **Throw**; I'm beginning to think a **Throw** is just an **int**. However, we don't need to consider that yet, since we can write **Frame.add** to take an **int**	2	e
15	K: Then I think we should do that for no other reason than it's simple. M: OK, this compiles and fails the test. Now, let's make the test pass. M: This compiles and passes the tests. But it's clearly simplistic. What's the next test case.	3	a
16	**M: Frame.add** is a fragile function. What if you call it with an 11? K: It can throw an exception if that happens. But who is calling it? Is this going to be an application framework that thousands of people will use and we have to protect against such things, or is this going to be used by you and only you? If the latter, just don't call it with an 11	4	a
17	M: the tests in the rest of the system will catch an invalid argument. If we run into trouble, we can put the check in later. So, the add function doesn't currently handle strikes or spares.	5	a
18	M: Let's write a test case that expresses that. K: if we call **add(10)** to represent a strike, what should **getScore** return? I don't know how to write the assertion, so maybe we're asking the wrong question. Or we're asking the right question to the wrong object.	5	a
19	M: When you call **add(10)**, or **add(3)** followed by **add(7)**, then calling **getScore** on the **Fram**e is meaningless. The frame would have to look ahead at later frames to calculate its score. If those later frames don't exist, then it would have to return something ugly like -1. I don't want to return -1	4	a
20	K: You've introduced the idea of frames knowing about other frames. Who is holding these different frame objects? So **Game** depends on **Frame**, and **Frame** in turn depends on **Game**	2	a

21	**M: Frame**s don't have to depend upon **Game**; they could be arranged in a linked list. Each frame could hold pointers to its next and previous frames. To get the score from a frame, the frame would look backwards to get the score of the previous frame and look forwards for any spare or strike balls it needs.	2	d
22	K: I'm feeling kind of dumb because I can't visualize this. Show me some code M: So, we need a test case first K: **Game** or another test for **Frame**? M: I think we need one for **Game**, since it's **Game** that will build the frames and hook them up to each other." K: Do you want to stop what we're doing on **Frame** and do a mental long jump to **Game**, or do you just want to have a **MockGame** object that does just what we need to get **Frame** working? M: let's stop working on **Frame** and start working on **Game**.	2	a
23	M: The test cases in **Game** should prove that we need the linked list of **Frame**s	5	a
24	K: but I'm still looking for proof for this list of **Frame**s. M: Let's keep following these test cases and see where they lead. M: OK, this compiles and fails the test. Now let's make it pass. M: This passes. Good	5	a
25	K: I'm still looking for this great proof of the need for a linked list of frame objects. That's what led us to **Game** in the first place."	3	a
26	M: I fully expect that once we start injecting spare and strike test cases, we'll have to build frames and tie them together in a linked list. But I don't want to build that until the code forces us to.	4	a
27	K: Let's keep going in small steps on **Game**. What about another test that tests two throws but with no spare? M: Yep, that one passes	5	a
28	M: Now let's try four balls, with no marks. That will pass too.	3	a
29	K: I didn't expect this. We can keep adding throws, and we don't ever even need a **Frame**. But we haven't done a spare or a strike yet. Maybe that's when we'll have to make one M: That's what I'm counting on	4	d
30	K: I forgot that we have to be able to show the score in each frame. M: First let's make this test case fail by adding the **scoreForFrame** method to **Game.** **M:** this compiles and fails. Now, how do we make it pass?	5	a
31	K: We can start making frame objects. M: We could just create an array of integers in **Game**. Each call to **add** would append a new integer onto the array. Each call to **scoreForFrame** will just work forward through the array and calculate the score. M: that works	5	a
32	K: Why the magic number 21 M: That's the maximum possible number of throws in a game K: "**scoreForFrame** needs to be refactored to be more communicative.	4	r
33	K: But before we consider refactoring, let me ask another question: Is **Game** the best place for this method? In my mind, **Game** is violating Bertrand	4	a

	Meyer's SRP (Single Responsibility Principle). It is accepting throws *and* it knows how to score for each frame. What would you think about a **Scorer** object?"		
34	M: but there are side-effects in the **score+=** expression. They don't matter here because it doesn't matter which order the two **addend** expressions are evaluated in. K: I suppose we could do an experiment to verify that there aren't any side-effects, but that function isn't going to work with spares and strikes. Should we keep trying to make it more readable or should we push further on its functionality?"	5	d
35	M: The experiment would only have meaning on certain compilers. Other compilers might use different evaluation orders. Let's get rid of the order dependency and then push on with more test cases.	4	r
36	M: next test case. Let's try a spare M: Let's refactor the test and put the creation of the game in a **setUp** function.	4	r
37	M: That's better now. let's write the spare test case. M: but I think the increment of ball in the **frameScore==10** case shouldn't be there. Here's a test case that proves my point	5	a
38	M: See, that fails. Now if we just take out that pesky extra increment. it still fails.... Could it be that the **score** method is wrong?	4	d
39	M: I'll test that by changing the test case to use **scoreForFrame(2)**	2	a
40	M: That passes. The **score** method must be messed up.	2	a
41	M: that's wrong. The **score** method is just returning the sum of the pins, not the proper score. What we need **score** to do is call **scoreForFrame** with the current frame	4	a
42	K: We don't know what the current frame is. Let's add that message to each of our current tests, one at a time	4	a
43	M: OK, that works. But it's stupid. Let's do the next test case	2	a
44	M: let's try the next. This one fails. Now let's make it pass	2	a
45	K: I think the algorithm is trivial. Just divide the number of throws by two, since there are two throws per frame. Unless we have a strike ... but we don't have strikes yet, so let's ignore them here too. K: What if we don't calculate it each time? What if we adjust a **currentFrame** member variable after each throw? M: OK, this works	4	a
46	M: But it also implies that the current frame is the frame of the last ball thrown, not the frame that the next ball will be thrown into	2	d
47	K: But before we go screwing around with it some more, let's pull that code out of **add** and put it in a private member function called **adjustCurrentFrame** or something	5	r
48	M: Now let's change the variable and function names to be more clear. What should we call **itsCurrentFrame**?	4	r
49	K: I kind of like that name. I don't think we're incrementing it in the right place though. The current frame, to me, is the frame number that I'm	4	a

	throwing in. So it should get incremented right after the last throw in a frame.		
50	M: Let's change the test cases to reflect that; then we'll fix **adjustCurrentFrame.** M: OK, that's working.	3	r
51	M: Now let's test **getCurrentFrame** in the two spare cases. This works.	3	a
52	M: Now, back to the original problem. We need **score** to work. We can now write **score** to call **scoreForFrame(getCurrentFrame()-1** M: This fails the **TestOneThrow** test case.	3	a
53	M: Let's look at it. With only one throw, the first frame is incomplete. The **score** method is calling **scoreForFrame(0)**. K: Who are we writing this program for, and who is going to be calling **score**? Is it reasonable to assume that it won't get called on an incomplete frame? M: To get around this, we have taken the **score** out of the **testOneThrow** test case. Is that what we want to do	4	r
54	K: We could even eliminate the entire **testOneThrow** test case. It was used to ramp us up to the test cases of interest. Does it really serve a useful purpose now? We still have coverage in all of the other test cases	3	e
55	M: Now, we'd better work on the strike test case. After all, we want to see all those **Frame** objects built into a linked list, don't we? M: this compiles and fails as predicted.	4	a
56	K: Now we need to make it pass. OK, that wasn't too hard.	2	a
57	M: Let's see if it can score a perfect game. M: it's saying the score is 330. K: Because the current frame is getting incremented all the way to 12. M: We need to limit it to 10.	3	a
58	M: now it's saying that the score is 270. What's going on? K: the **score** function is subtracting one from **getCurrentFrame**, so it's giving you the score for frame 9, not 10. M: You mean I should limit the current frame to 11 not 10?	4	a
59	M: OK, so now it gets the score correct, but fails because the current frame is 11 and not 10.	2	a
60	M: We want the current frame to be the frame the player is throwing into, but what does that mean at the end of the game? K: Maybe we should go back to the idea that the current frame is the frame of the last ball thrown M: Or maybe we need to come up with the concept of the last *completed* frame? After all, the score of the game at any point in time is the score in the last completed frame. K: A completed frame is a frame that you can write the score into. Yes, a frame with a spare in it completes after the next ball. M: A frame with a strike in it completes after the next two balls. A frame with no mark completes after the second ball in the frame.	3	a
61	M: We are trying to get the **score** method to work, right? All we need to do is force **score** to call **scoreForFrame(10)** if the game is complete. K: How do we know if the game is complete?"	3	a

	M: If **adjustCurrentFrame** ever tries to increment **itsCurrentFrame** past the 10th frame, then the game is complete.		
62	K: All you are saying is that if **getCurrentFrame** returns 11, the game is complete; that's the way the code works now. M: You mean we should change the test case to match the code. M: that works.	2	a
63	M: I suppose it's no worse than **getMonth** returning zero for January, but I still feel uneasy about it. K: That doesn't fail.	3	a
64	K: I thought since the 21st position of the array was a strike, the scorer would try to add the 22nd and 23rd positions to the score. But I guess not.	2	a
65	M: you are still thinking about that **scorer** object, aren't you? Anyway, I see what you were getting at, but since **score** never calls **scoreForFrame** with a number larger than 10, the last strike is not actually counted as a strike. It's just counted at a 10 to complete the last spare. We never walk beyond the end of the array. K: , let's pump our original score card into the program K: Well, that works. Are there any other test cases that you can think of	4	a
66	M: let's test a few more boundary conditions. How about the poor schmuck who throws 11 strikes and then a final 9. M: That works.	4	a
67	M: how about a 10th frame spare? M: That works too.	4	a
68	K: Besides I really want to refactor this mess. I still see the **scorer** object in there somewhere. M: I'd really like to extract the body of that else clause into a seperate function named **handleSecondThrow**, but I can't because it uses **ball**, **firstThrow**, and **secondThrow** local variables. K: We could turn those locals into member variables. M: that kind of reinforces your notion that we'll be able to pull the scoring out into its own **scorer** object.	3	r
69	K: I hadn't expected the name collision. We already had an instance variable named **firstThrow**. But it is better named **firstThrowInFrame**. Anyway, this works now. So we can pull the else clause out into its own function	4	r
70	M: Look at the structure of **scoreForFrame**! In pseudocode, it looks something like this. K: That's pretty much the rules for scoring bowling isn't it? let's see if we can get that structure in the real function. First let's change the way the **ball** variable is being incremented so that the three cases manipulate it independently.	3	r
71	M: now let's get rid of the **firstThrow** and **secondThrow** variables and replace them with appropriate functions.	2	r
72	M: That step works; let's keep going	2	a
73	M: OK, that works too. Now let's deal with **frameScore**	2	a
74	K: you aren't incrementing **ball** in a consistent manner. In the spare and strike case, you increment before you calculate the score. In the **twoBallsInFrame** case, you increment *after* you calculate the score. And	3	r

	the code *depends* upon this order! M: I'm planning on moving the increments into **strike**, **spare**, and **twoBallsInFrame**. That way they'll disappear from the **scoreForFrame** function, and the function will look just like our pseudocode		
75	M: OK, now since nobody uses **firstThrow**, **secondThrow**, and **frameScore** anymore, we can get rid of them.	3	e
76	M: since the only variable that couples the three cases is **ball**, and since **ball** is dealt with independently in each case, we can merge the three cases together	3	r
77	M: you would like to move the increments? M: Look at that **scoreForFrame** function. That's the rules of bowling stated about as succinctly as possible.	2	a
78	K: what happened to the linked list of **Frame** objects?	2	a
79	K: We made a mistake starting with the **Throw** class. We should have started with the **Game** class first M: , next time let's try starting at the highest level and work down K: Down Design!??!?! Could DeMarco have been right all along. M: Correction: Top Down *Test First* Design. Frankly, I don't know if this is a good rule or not. It's just what would have helped us in this case. So next time, I'm going to try it and see what happens.	4	a
80	K: Anyway we still have some refactoring to do. The **ball** variable is just a private iterator for **scoreForFrame** and its minions. They should all be moved into a different object M: your **Scorer** object. You were right after all. Let's do it	3	r
81	K: That's much better. Now **Game** just keeps track of frames, and **Scorer** just calculates the score. The SRP rocks. M: Did you notice that the **itsScore** variable is not being used anymore? K: You're right. Let's kill it	4	e
82	K: Now should we clean up the **adjustCurrentFrame** stuff. M: first let's extract the increments into a single function that also restricts the frame to 11.	3	r
83	M: that's a little better. Now let's break out the strike case into its own function. M: That -1 is odd. It's the only place we truly use **getCurrentFrame**, and yet we need to adjust what it returns. M: The code wants **itsCurrentFrame** to represent the frame of the last thrown ball, not the frame we are about to throw into. M: we should remove **getCurrentFrame** from all the test cases and remove the **getCurrentFrame** function itself. Nobody really uses it K: , I get your point. I'll do it. It'll be like putting a lame horse out of its misery. M: You mean to tell me that we were fretting over *that*. All we did was change the limit from 11 to 10 and remove the -1. K: it really wasn't worth all the angst we gave it	4	r
84	M: looks like we are done. Let's just read through the whole program and see if it's as simple and communicative as it can be	2	a

APPENDIX B

CASE STUDY ON INCREMENTAL SOFTWARE DEVELOPMENT

In this section, we include those detailed data for each intermediate pair during incremental software development. Tables B.1-B.8 show the distribution of the cognitive activities and Bloom's levels throughout the recorded episodes for pair I to pair VIII. Figures B.1-B.8 are the UML class diagrams for the programs developed by pair I to pair VIII. The data are summarized in Chapter 7.

Table B.1: The distribution of the cognitive activities and Bloom's levels throughout the recorded episodes for pair I

	Assimilation		Accommodation		Total
	Absorption	Denial	Reorganization	Expulsion	
Recognition	0 0%	0 0%	0 0%	0 0%	0 0%
Comprehension	18 27.69%	0 0%	2 3.08%	0 0%	20 30.77%
Application	19 29.23%	0 0%	3 4.62%	1 1.54%	23 35.39%
Analysis	8 12.31%	0 0%	0 0%	2 3.07%	10 15.38%
Synthesis	12 18.46%	0 0%	0 0%	0 0%	12 18.46%
Evaluation	0 0%	0 0%	0 0%	0 0%	0 0%
Total	57 87.69%	0 0%	5 7.70%	3 4.61%	65 100%

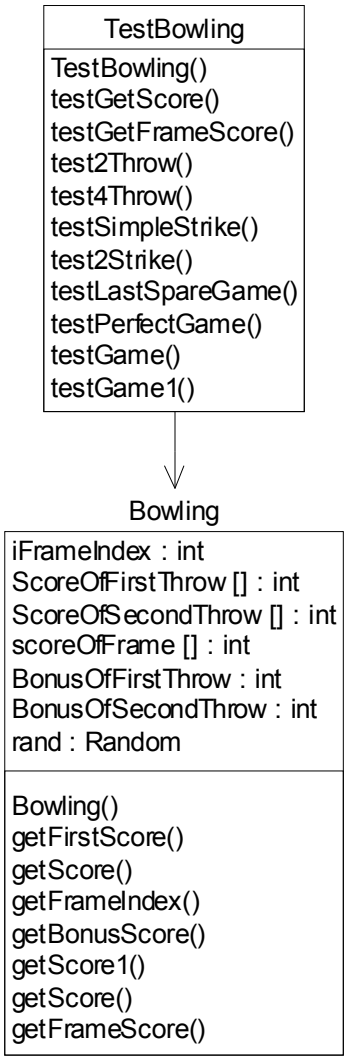

Figure B.1: UML class diagram for the pair I's program

Table B.2: The distribution of the cognitive activities and Bloom's levels throughout the recorded episodes for pair II

	Assimilation		Accommodation		Total
	Absorption	Denial	Reorganization	Expulsion	
Recognition	0 0%	0 0%	0 0%	0 0%	0 0%
Comprehension	14 31.11%	0 0%	0 0%	0 0%	14 31.11%
Application	17 37.78%	1 2.22%	2 4.45%	0 0%	20 44.45%
Analysis	6 13.33%	0 0%	0 0%	0 0%	6 13.33 %
Synthesis	5 11.11%	0 0%	0 0%	0 0%	5 11.11%
Evaluation	0 0%	0 0%	0 0%	0 0%	0 0%
Total	42 93.33%	1 2.2%	2 4.45%	0 0%	45 100%

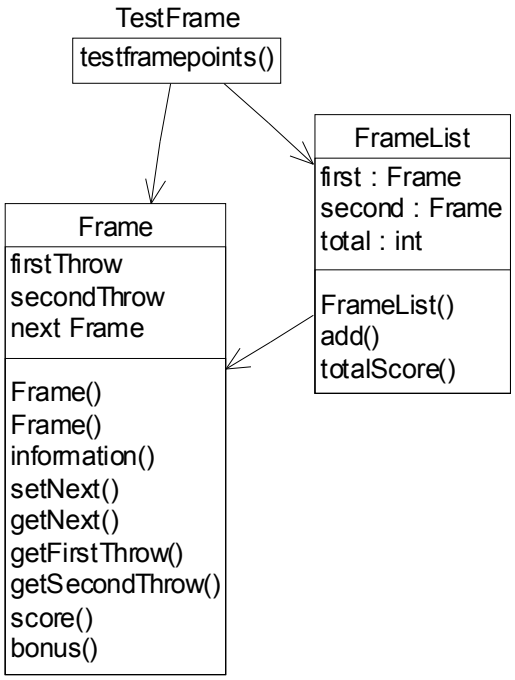

Figure B.2: UML class diagram for the pair II's program

Table B.3: The distribution of the cognitive activities and Bloom's levels throughout the recorded episodes for pair III

	Assimilation		Accommodation		Total
	Absorption	Denial	Reorganization	Expulsion	
Recognition	0 0%	0 0%	0 0%	0 0%	0 0%
Comprehension	12 19.35%	1 1.61%	0 0%	0 0%	13 20.96%
Application	15 24.20%	0 0%	7 11.29%	0 0%	22 35.49%
Analysis	13 20.97%	0 0%	2 3.23%	0 0%	15 24.20%
Synthesis	12 19.35%	0 0%	0 0%	0 0%	12 19.35%
Evaluation	0 0%	0 0%	0 0%	0 0%	0 0%
Total	52 83.87%	1 1.61%	9 14.52%	0 0%	62 100%

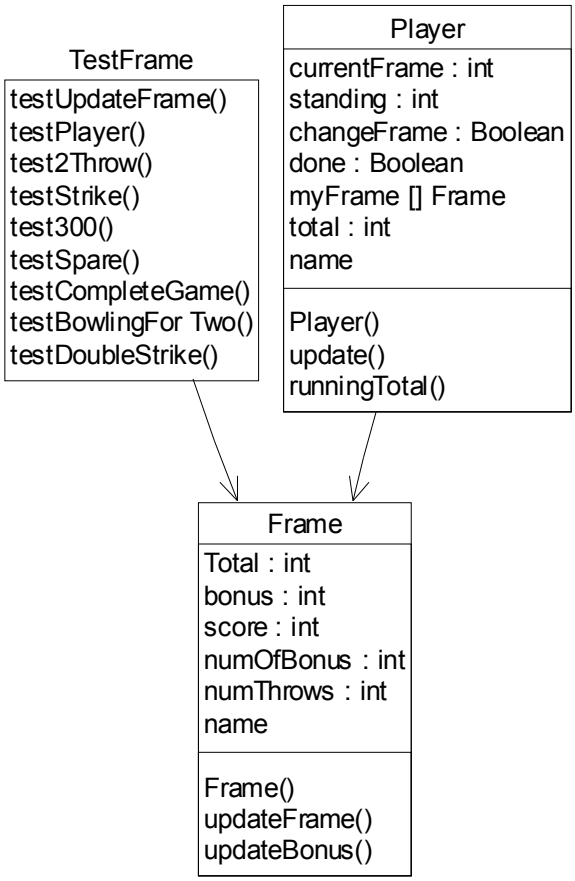

Figure B.3: UML class diagram for the pair III's program

Table B.4: The distribution of the cognitive activities and Bloom's levels throughout the recorded episodes for pair IV

	Assimilation		Accommodation		Total
	Absorption	**Denial**	**Reorganization**	**Expulsion**	
Recognition	0 0%	0 0%	0 0%	0 0%	0 0%
Comprehension	15 21.54%	0 0%	1 1.45%	0 0%	16 23.19%
Application	26 37.68%	0 0%	1 1.45%	1 1.45%	28 40.58%
Analysis	10 14.49%	0 0%	1 1.45%	0 0%	11 15.94%
Synthesis	14 20.29%	0 0%	0 0%	0 0%	14 20.29%
Evaluation	0 0%	0 0%	0 0%	0 0%	0 0%
Total	65 94.20%	0 0%	3 4.35%	1 1.45%	69 100%

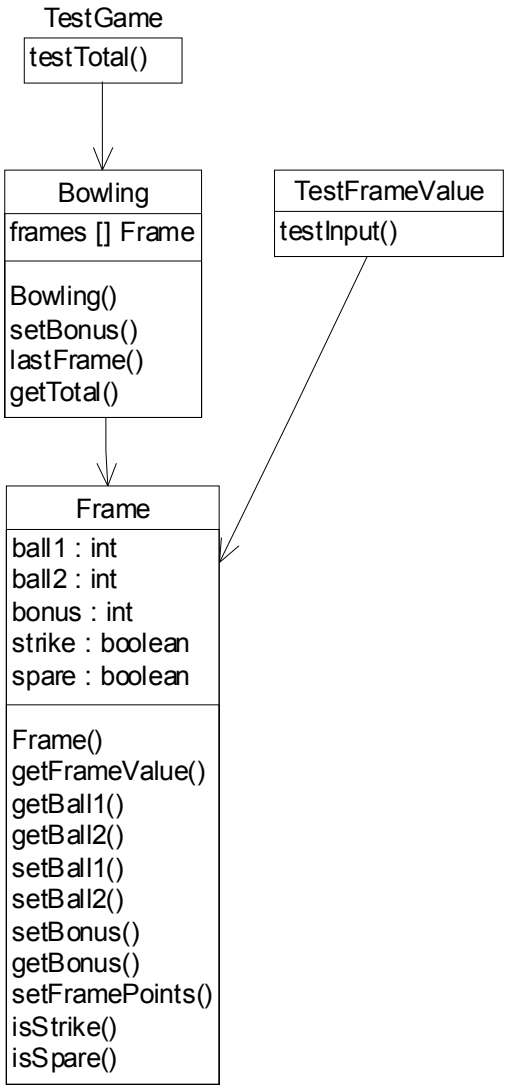

Figure B.4: UML class diagram for the pair IV's program

Table B.5: The distribution of the cognitive activities and Bloom's levels throughout the recorded episodes for pair V

	Assimilation		Accommodation		Total
	Absorption	Denial	Reorganization	Expulsion	
Recognition	0 0%	0 0%	0 0%	0 0%	0 0%
Comprehension	16 23.88%	0 0%	0 0%	0 0%	16 23.88%
Application	25 37.31%	0 0%	0 3.00%	1 1.49%	26 38.81%
Analysis	12 17.91%	0 0%	2 3%	0 0%	14 20.90%
Synthesis	11 16.41%	0 0%	0 0%	0 0%	11 16.41%
Evaluation	0 0%	0 0%	0 0%	0 0%	0 0%
Total	64 95.51%	0 0%	2 3.00%	1 1.49%	67 100%

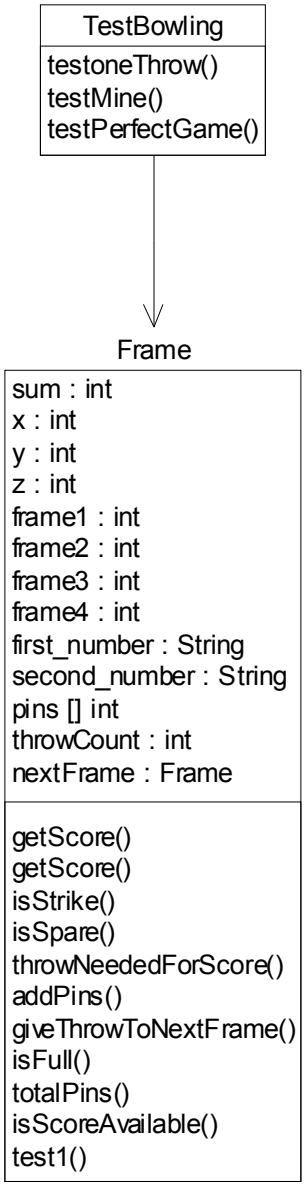

Figure B.5: UML class diagram for the pair V's program

Table B.6: The distribution of the cognitive activities and Bloom's levels throughout the recorded episodes for pair VI

	Assimilation		Accommodation		Total
	Absorption	Denial	Reorganization	Expulsion	
Recognition	0 0%	0 0%	0 0%	0 0%	0 0%
Comprehension	11 16.66%	0 0%	1 1.52%	1 1.52%	13 19.70%
Application	23 34.85%	0 0%	1 1.52%	1 1.52%	25 37.88%
Analysis	13 19.70%	0 0%	1 1.52%	0 0%	14 21.21%
Synthesis	14 21.21%	0 0%	0 0%	0 0%	14 21.21%
Evaluation	0 0%	0 0%	0 0%	0 0%	0 0%
Total	61 92.42%	0 0%	3 4.55%	2 3.03%	66 100%

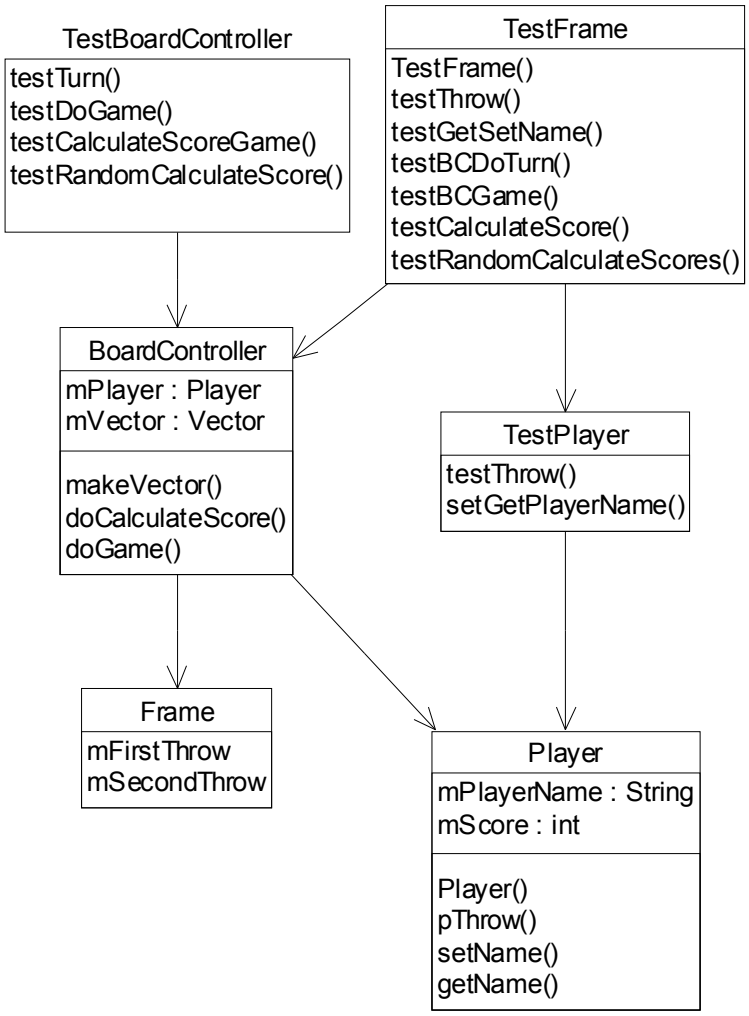

Figure B.6: UML class diagram for the pair VI's program

Table B.7: The distribution of the cognitive activities and Bloom's levels throughout the recorded episodes for pair VII

	Assimilation		Accommodation		Total
	Absorption	Denial	Reorganization	Expulsion	
Recognition	0 0%	0 0%	0 0%	0 0%	0 0%
Comprehension	19 29.69%	0 0%	0 0%	0 0%	19 29.69%
Application	24 37.5%	0 0%	0 0%	1 1.56%	25 39.06%
Analysis	7 10.94%	0 0%	0 0%	0 0%	7 10.94 %
Synthesis	13 20.31 %	0 0%	0 0%	0 0%	13 20.31%
Evaluation	0 0%	0 0%	0 0%	0 0%	0 0%
Total	63 98.44%	0 0%	0 0%	1 1.56%	64 100%

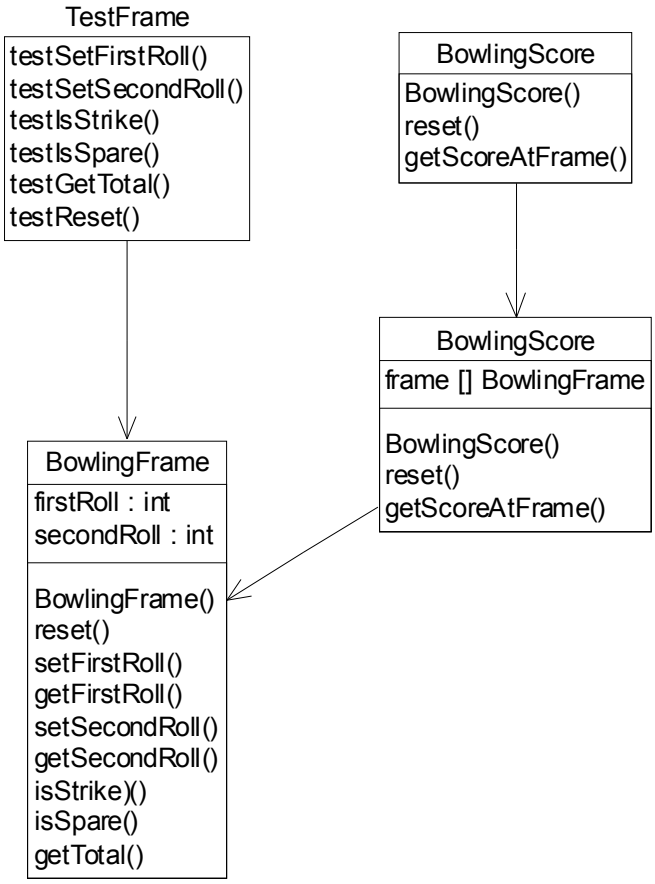

Figure B.7: UML class diagram for the pair VII's program

Table B.8: The distribution of the cognitive activities and Bloom's levels throughout the recorded episodes for pair VIII

	Assimilation		Accommodation		Total
	Absorption	Denial	Reorganization	Expulsion	
Recognition	0 0%	0 0%	0 0%	0 0%	0 0%
Comprehension	15 25.00%	0 0%	0 0%	0 0%	15 25.00%
Application	22 36.67%	0 0%	2 3.33%	1 1.67%	25 41.67%
Analysis	8 13.33%	0 0%	0 0%	0 0%	8 13.33%
Synthesis	12 20.00%	0 0%	0 0%	0 0%	12 20.00%
Evaluation	0 0%	0 0%	0 0%	0 0%	0 0%
Total	57 95.00 %	0 0%	2 3.33%	1 1.67%	60 100%

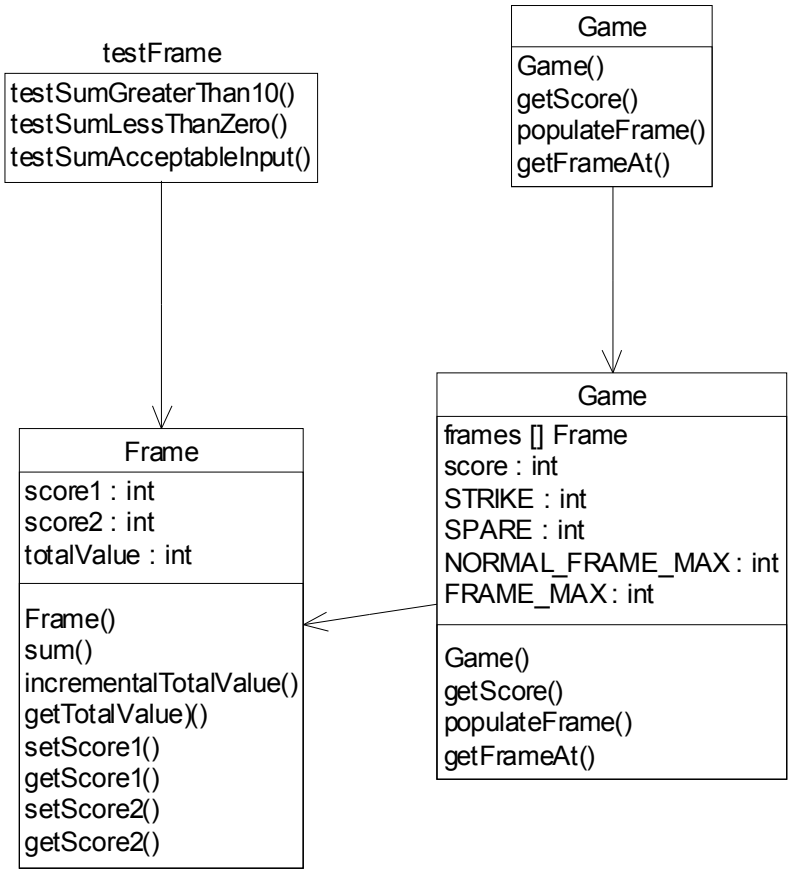

Figure B.8: UML class diagram for the pair VIII's program

APPENDIX C

CASE STUDY ON PROGRAM DEBUGGING

The case study description and the distribution of the cognitive activities and Bloom levels are in Chapter 6. The Table C.1 contains the classification of cognitive activities and Bloom's levels for all the 28 episodes.

Table C.1: Classification of the cognitive activities and Bloom's levels in the case study on program debugging
(A: absorption; d: denial; r: reorganization; e: expulsion; 1-6 refers to the six Bloom's levels starting from knowledge)

Episode	Programmer's action	Bloom level	Cognitive activity
1	The programmer was provided with the symptoms "system not responding or too slow".	2	a
2	Based on his experience with the architecture of similar systems, the programmer considered several possible causes.	4	a
3	He decided that the most likely cause is the slow database query that slows down access to the web server. The slow web server then blocks access to the site.	5	r
4	Based on this assumption, the programmer used a "browser skeleton" to validate this hypothesis.	4	a
5	The browser skeleton is a test harness, in which a program simulates customers connecting to the web server.	3	a
6	The average response time was recorded after a browser skeleton was launched.	2	a
7	Then the actual web browser was launched to connect to the server and the response time of the server was also recorded.	2	a
8	The result turned out to be 100 times longer than the response time for the browser skeleton after contrasting the two response times. This test was repeated several times to make sure that the results were stable and repeatable.	4	a
9	The test results supported the hypothesis that queries to the database slow down access to the web server.	5	a
10	The programmer has narrowed down the cause of the problem, but further work is needed.	6	a

11	The next hypothesis was that the web server could not spawn child processes to handle customer requests.	5	r
12	Based on this assumption, another test was conducted using the same test harness to observe the state of all processes.	2	a
13	If the hypothesis were true, the number of processes should not increase after a certain period of time.	3	r
14	However, the result did show that some new processes were spawned, and the overall number of processes increased.	6	a
15	Therefore, the second hypothesis was not supported by the evidence.	1	e
16	Nonetheless, a useful observation was obtained: One of the child processes was accumulating CPU time and others were not.	2	a
17	Based on this observation, the second hypothesis was modified as "not all child processes were handling inbound requests"	5	r
18	A further investigation was conducted to see what each child was doing during the browser request process.	3	a
19	It was found that only one child works properly and other children tried to set a lock on a file that was already locked by another process.	4	a
20	Therefore, there was a block or race condition that put other child processes into an infinite loop and blocked their execution.	6	r
21	After reading the Unix manual and going through documentation and configuration settings, the programmer noticed that the web server uses system calls getcontext and setcontext, which are used in user-level threaded applications.	2	a
22	The programmer knew that the web server was actually a multi-threaded application with user-level threads and found that the child process was not threading as expected.	3	a
23	Therefore a further hypothesis was made that the actual COTS server agent was not multi-threaded and, forced the web server to become blocked.	5	r
24	The server agent provides communication between the web server and the relational database management system. The present server agent provided by the database vendor is used with the www server-side API although it also supports the Common Gateway Interface (CGI).	2	a
25	If the above hypothesis is true, then using the CGI version of the same agent instead of www server-side API version of the server agent would be the right solution, as CGI supports multiple threading.	6	r
26	A new test was proposed after the www server-side API version of the server agent was replaced with CGI version.	3	a
27	The test harness and actual web browser were used again.	1	a
28	The result demonstrated that the web server was no longer blocked after the replacement.	2	r

BIBLIOGRAPHY

[1] Adair, G., "The Hawthorne effect: a reconsideration of the methodological artifact", *Journal of Applied Psychology*, vol. 69, no. 2, 1984, pp. 334-345.

[2] Alavi, M. and Carlson, P., "A review of MIS research and disciplinary development", *Journal of Management Information Systems*, vol. 8, no. 4, 1992, pp. 45-62.

[3] Araki, K., Furukawa, Z., and Cheng, J., "A general framework for debugging", *IEEE Software* May 1991, pp. 14-20.

[4] Atwood, M. E. and Ramsey, H. R., "Cognitive structures in the comprehension and memory of computer programs: an investigation of computer program debugging." US Army Research Institute for the Behavioral and Social Sciences, Alexandra,VA 1978.

[5] Baecker, R., DiGiano, C., and Marcus, A., "Software visualization for debugging", *Communications of the ACM*, vol. 40, no. 4, April 1997, pp. 44–54

[6] Baniassad, E. and Murphy, G., "Conceptual module querying for software reengineering", in Proceedings of 20th International Conference on Software Engineering, Kyoto, Japan, 1998, pp. 64-73.

[7] Baragry, J., *Understanding Software Engineering: From Analogies with other Disciplines to Philosophical Foundations*, Ph.D. Dissertation, La Trobe University, Bundoora, Australia, 2000.

[8] Baragry, J. and Reed, K., "Why we need a different view of software architecture", in Proceedings of the Working IEEE/IFIP Conference on

Software Architecture (WICSA01), Amsterdam, the Netherlands, 2001, pp. 125.

[9] Basili, V. R., "The role of experimentation in software engineering: past, present, future ", in Proceedings of the 18th International Conference on Software Engineering, Los Alamitos, CA, 1996, pp. 442-449.

[10] Beck, K., *Extreme Programming Explained*, Maassachusettes, Addison-Wesley, 2000.

[11] Benbunan-Fich, R., "Using protocol analysis to evaluate the usability of a commercial web site", *Information & Management*, vol. 39, 2001, pp. 151-163.

[12] Benedusi, P., Benvenuto, V., and Tomacelli, L., "The role of testing and dynamic analysis in program comprehension supports", in Proceedings of the IEEE Second Workshop on Program Comprehension, 1993, pp. 149-158.

[13] Berlin, L. M., "Beyond program understanding: a look at programming expertise in industry", in *Empirical Studies of Programmers: Fifth Workshop*, Cook, C. R., Scholtz, J. C., and Spohrer, J. C., Eds., Norwood, NJ Ablex Publishing, 1993, pp. 8-25.

[14] Berry, D. C. and Boardbent, D. E., "The role of instruction and verbalization in improving performance on complex search tasks", *Behavior and Information Techonology*, vol. 9, no. 3, 1990, pp. 175-190.

[15] Biederman, G. B., Stepaniuk, S., Davey, V. A., Raven, K., and Ahn, D., "Observational learning in children with down syndrome and developmental

delays: the effect of presentation speed in videotaped modelling", *Down Syndrome Research and Practice*, vol. 6, no. 1, 1999, pp. 12-18.

[16] Biggerstaff, T. J., Mitbander, B. G., and Webster, D. E., "Program understanding and the concept assignment problem", *Communication of the ACM*, vol. 37, no. 5, May 1994, pp. 72-82.

[17] Bisant, D. B. and Groninger, L., "Cognitive processes in software fault detection: a review and synthesis", *International Journal of Human-Computer Interaction*, vol. 5, no. 2, 1993, pp. 189-206.

[18] Bloom, B. S., "Taxonomy of Educational Objectives: The Classification of Educational Goals: Handbook, I, Cognitive Domain": New York, Toronto, Longmans, Green, 1956.

[19] Bohm, D. and Nichol, L., *On Dialogue*, Brunner-Routledge, 1996.

[20] Brereton, P., Lees, S., Gumbley, M., Boldyreft, C., Drummond, S., Layzell, P., Macaulay, L., and Young, R., "Distributed group working in software engineering education", *Information & Software Technology*, vol. 40, 1998, pp. 221-227.

[21] Brook, F., *The Mythical Man-Month*, Addison-Wesley, MA, 1975.

[22] Brooks, R., "Towards a theory of the cognitive processes in computer programming", *International Journal of Man-Machine Studies*, vol. 9, no. 6, 1977, pp. 737-742.

[23] Brooks, R., "Studying programmers behavior experimentally: the problems of proper methodology", *Communications of ACM*, vol. 23, no. 4, 1980, pp. 207-213.

[24] Brooks, R., "Towards a theory of the comprehension of computer programs", *International Journal of Man-Machine Studies*, vol. 18, no. 6, 1983, pp. 543-554.

[25] Buckley, J. and Exton, C., "Bloom's taxonomy: a framework for assessing programmers' knowledge of software systems", in Proceedings of the 11th International Workshop on Program Comprehension, Portland, Oregon, 2003, pp. 165-174.

[26] Burkhardt, J., DTtienne, F., and Wiedenbeck, S., "The effect of object-oriented programming expertise in several dimensions of comprehension strategies", in Proceedings of the 6th International Workshop on Program Comprehension, Ischia, Italy, 24 - 26 June 1998, pp. 82-89.

[27] Canfora, G., Cimitile, A., and Visaggio, C., "Working in pairs as a means for design knowledge building: an empirical study", in Proceedings of International Workshop on Program Comprehension, Bari, Italy, 24-26 June 2004, pp. 62-69.

[28] Chen, K. and Rajlich, V., "Case study of feature location using dependence graph", in Proceedings of the 8th International Workshop on Program Comprehension (IWPC'00), Limerick, Ireland, June 2000, pp. 241-249.

[29] Christensen, L. B., *Experimental Methodology*, Allyn and Bacon, 1994.

[30] Clayton, R., Rugaber, S., Taylor, L., and Wills, L., "A case study of domain-based program understanding", in Proceedings of 5th Workshop on Program Comprehension, Dearborn, Michigan, May 28-30 1997, pp. 102-110.

[31] Clayton, R., Rugaber, S., and Wills, L., "Dowsing: a tool framework for domain-oriented browsing of software artifacts", in Proceedings of the 13th IEEE International Conference on Automated Software Engineering (ASE '98), Honolulu, Hawaii, October 1998, pp. 204 -207.

[32] Clayton, R., Rugaber, S., and Wills, L., "On the knowledge required to understand a program", in Proceedings of the Fifth IEEE Working Conference on Reverse Engineering, Honolulu, Hawaii, October 1998, pp. 69-78.

[33] Clements, P., Krut, R., Morris, E., and Wallnau, K., "The Gadfly: an approach to architectural-level system comprehension", in Proceedings of Fourth IEEE Workshop on Program Comprehension, Berlin, 1996, pp. 178-186.

[34] Cockburn, A. and Williams, L., "The costs and benefits of pair programming", in Proceedings of Extreme Programming and Flexible Processes in Software Engineering, Cagliari, Italy, June 21-23 2000, pp. 223-243.

[35] Confrey, J., "Voice and perspective: hearing epistemological innovation in students' words", in *Revue des Sciences de l'education. Special Issue: Constructivism in Education*, vol. 20(1), Bednarz, N., Larochelle, M., and Desautels, J., Eds., 1994, pp. 115-133.

[36] Corritore, C. and Wiedenbeck, S., "Direction and scope of comprehension-related activities by Procedural and Object-Oriented programmers: an

empirical study", in Proceedings of 8th International Workshop on Program Comprehension (IWPC'00), Limerick, Ireland, 10 - 11 June 2000, pp. 139.

[37] Corritore, C. L. and Wiedenbeck, S., "What do novices learn during program comprehension?" *International Journal of Human-Computer Interaction*, vol. 3, no. 2, 1991, pp. 199-222.

[38] Corritore, C. L. and Wiedenbeck, S., "Mental representations of expert Procedural and Object-Oriented programmers in a software maintenance task", *International Journal of Human-Computer Studies*, vol. 50, no. 1, 1999, pp. 61-83.

[39] Cubranic, D., Murphy, G. C., Singer, J., and Booth, K. S., "Hipikat: a project memory for software development", *IEEE Transactions on Software Engineering*, vol. 31, no. 6, 2005, pp. 446 - 465.

[40] Curtis, B., "By the way, did anyone study any real programmers?" *Empirical Studies of Programmers, Ablex Publishing Corporation, Norwood,NJ* 1986, pp. 256-262.

[41] Curtis, B., "A field study of the software design process on large systems", *Communications of the ACM*, vol. 13, no. 11, 1988, pp. 1268-1287.

[42] Dalton, J. and Smitd, D., *Extending Children's Special Abilities: Strategies for Primary Classrooms*, Melbourne, Victorian Ministry of Education, 1986.

[43] Davies, S. P., "The nature and development of programming plans", *International Journal of Man-Machine Studies*, vol. 32, no. 4, 1990, pp. 461-481.

[44] Davies, S. P., "Externalising Information During Coding Activities: Effects of Expertise, Environment and Task", in *Empirical Studies of Programmers: Fifth Workshop.*, C. R. Cook, J. C. Scholtz, and Spohrer, J. C., Eds., Norwood, NJ Ablex Publishing, 1993, pp. 42-61.

[45] Davies, S. P., "Models and theories of programming strategy", *International Journal of Man-Machine Studies*, vol. 39, no. 2, 1993, pp. 237-267.

[46] Davis, J. S., "Chunks: a basis for complexity measurement", *Information Processing & Mgt.*, vol. 20, no. 1-2, 1984, pp. 119-127.

[47] Detienne, F., "Difficulties in designing with an object-oriented language: an empirical study", *D. Diaper,D. Gilmore,G.Cockton, and B.Shacker (eds), Human Computer Interaction, Proceedings of INTERACT'90, North Holland* 1990, pp. 971-976.

[48] Driscoll, M. P., *Psychology of Learning for Instruction*, Allyn and Bacon, 2000.

[49] Ducasee, M., "A pragmatic survey of automatic debugging", in Proceedings of 1st International Workshop on Automated and Algorithmic Debugging, Linköping, Sweden, May 3-5 1993, pp. 1-15.

[50] Duncker, K., *On problem solving*, Washington, The American Psychological Association, 1945.

[51] Ericsson, K. and Simon, H. A., *Protocol Analysis: Verbal Reports as Data*, Cambridge, Mass, MIT Press, 1993.

[52] Exton, C., "Constructivism and program comprehension strategies", in Proceedings of the 10th International Workshop on Program Comprehension, Paris, France, 27-29 June 2002, pp. 281-284.

[53] Fischer, G., McCall, R., Ostwald, J., Reeves, B., and Shipman, F., "Seeding, evolutionary growth and reseeding: supporting the incremental development of design environments", in Proceedings of the Conference on Computer-Human Interaction (CHI'94), Boston, MA, 1994, pp. 292-298.

[54] Fischer, G. and Ostwald, J., "Knowledge management: problems, promises, realities, and challenges", *IEEE Intelligent Systems*, vol. 16, no. 1, January/February 2001, pp. 60-72.

[55] Fix, V., Wiedenbeck, S., and Scholtz, J., "Mental representations of programs by novices and experts", in Proceedings of the Conference on Human Factors and Computing Systems (INTERCHI'93), Amsterdam The Netherlands, April 24-29 1993, pp. 74-79.

[56] Fowler, M., *UML Distilled*, Addison Wesley Longman, 1997.

[57] Fowler, M., *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 1999.

[58] Frohmann, B., "Discourse analysis as a research method in library and information science", *Library and Information Science Research* no. 16, 1994, pp. 119-138.

[59] Fu, W., "ACT-PRO: action protocol tracer -- a tool for analyzing discrete action protocols ", *Behavior Research Methods, Instruments, & Computers*, vol. 33, no. 2, 2001, pp. 149-158

[60] Gagnon, G. J. and Collay, M., *Design for Learning, Six Elements in Constructivist Classrooms*, Corwin Press Inc., 2001.

[61] Gallis, H., Arisholm, E., and Dyba, T., "A transition from partner programming to pair programming - an industrial case study", in Proceedings of Pair Programming Workshop in 17th annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), Seatte, USA, 2002, pp. position paper.

[62] Gallis, H., Arisholm, E., and Dyba, T., "An initial framework for research on pair programming", in Proceedings of the Second International Symposium on Empirical Software Engineering, Rome, Italy, September 30 - October 1 2003, pp. 132-142.

[63] Gamma, E. and Kent, B., *Contributing to Eclipse: Principles, Patterns, and Plugins*, Addison-Wesley 2003.

[64] Gardner, H., *Frames of Mind: The Theory of Multiple Intelligences*, Basic Books, 1993.

[65] Gero, J. S. and McNeill, T. M., "An approach to the analysis of design protocols", *Design Studies*, vol. 19, no. 1, 1998, pp. 21-61.

[66] Gilmore, D. J., "Expert programming knowledge: a strategic approach", *Psychology of Programming* 1990, pp. 223-233.

[67] Glasser, W., *The Quality School*, Perennial, 1998.

[68] Gnatz, M., Kof, L., Prilmeier, F., and Seifert, T., "A practical approach of teaching software engineering", in Proceedings of 16th Conference on

Software Engineering Education and Training, Madrid, Spain, March 20-22 2003, pp. 140-147.

[69] Gould, J. D., "Some psychological evidence on how people debug computer programs", *International Journal of Man-machine Studies*, vol. 7, no. 1, 1975, pp. 151-181.

[70] Gray, W. D. and Anderson, J. R., "Change episodes in coding: when and how do programmers change their code?" in *Empirical Studies of Programmers*, Olson, G. M., Sheppard, S., and Soloway, E., Eds., Norwood, NJ Ablex Publishing Corporation, 1987, pp. 185-197.

[71] Gruber, T., "A translation approach to portable ontologies", *Knowledge Acquisition*, vol. 5, no. 2, 1993, pp. 199-220.

[72] Gugerty, L. and Olson, G. M., "Debugging by skilled and novice programmers", in Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, Boston, Massachusetts, United States, 1986, pp. 171-174.

[73] Hedin, G., Bendix, L., and Magnusson, B., "Introducing software engineering by means of Extreme Programming", in Proceedings of International Conference on Software Engineering, Portland, Oregon May 3-10 2003, pp. 586 - 593.

[74] Henninger, S., "Tools supporting the creation and evolution of software development knowledge", in Proceedings of the International Conference on Automated Software Engineering (ASE'97), Incline Village, Nevada, 1997, pp. 46-53.

[75] Hissam, S., "Case study: correcting system failure in a COTS information system, SET monographs on the use of commercial software in government systems", Technical Report, Carnegie Mellon University 1997.

[76] Hollan, J., Hutchins, E., and Kirsch, D., "Distributed cognition: toward a new foundation for human-computer interaction research", *ACM Transactions on Computer-Human Interaction*, vol. 7, no. 2, 2000, pp. 174-196.

[77] Holt, R. W., Boehm-Davis, D. A., and Schultz, A. C., "Mental representations of programs for student and professional programmers", in *Empirical Studies of Programmers: Second Workshop*, Olson, G. M., Sheppard, S., and Soloway, E., Eds., E Norwood, NJ Ablex Publishing Co., 1987, pp. 33-46.

[78] Horowitz, S., Reps, T., and Binkley, D., "Interprocedural slicing using dependence graphs", *ACM Transactions on Programming Languages and Systems* vol. 12, no. 1, January 1990, pp. 26-60.

[79] Huitt, W., "Bloom et al.'s taxonomy of the cognitive domain: educational psychology interactive", Valdosta, GA: Valdosta State University, http://chiron.valdosta.edu/whuitt/col/cogsys/bloom.html, 2004.

[80] IEEE-CS and ACM, "Software Engineering Education Knowledge (SEEK): Second Draft, available at http://sites.computer.org/ccse, Dec", 2002.

[81] Javadoc, "Javadoc website", http://java.sun.com/j2se/javadoc, 2006.

[82] Jensen, E., *Brain-based Learning: The New Science of Teaching and Training*, Revision Edition ed., Brain Store Inc, 2000.

[83] Kitchenham, B. A., Pfleeger, S. L., Pickard, L. M., Jones, P. M., Hoaglin, D. C., Eman, K. E., and Rosenberg, J., "Preliminary guidelines for empirical

research in software engineering", *IEEE Transactions on Software Engineering*, vol. 28, no. 8, 2002, pp. 721-734.

[84] Ko, A. J. and Myers, B. A., "Development and evaluation of a model of programming errors", in Proceedings of IEEE Symposia on Human-Centric Computing Languages and Environments, Auckland, New Zealand, October 2003, pp. 7-14.

[85] Koenemann, J. and Robertson, S., "Expert problem solving strategies for problem comprehension", in Proceedings of the Conference on Human Factors and Computing Systems (CHI'91), New Orleans, LA, April 27 - May 2 1991, pp. 125-130.

[86] Kozaczynski, W., Ning, J., and Engberts, A., "Program concept recognition and transformation", *IEEE Transactions on Software Engineering*, vol. 18, no. 12, 1992, pp. 1065-1075.

[87] Larman, C. and Basili, V., "Iterative and incremental development: a brief history", *IEEE Computer* June 2003, pp. 47-56.

[88] Lehman, M., Perry, D., and Ramil, J., "Implications of evolution metrics on software maintenance", in Proceedings of International Conference on Software Maintenance, Bethesda, Maryland, 1998, pp. 208.

[89] Letovsky, S., "Cognitive Processes in Program Comprehension", in *Empirical Studies of Programmers*, Albex, 1986, pp. 58-79.

[90] Letovsky, S. and Soloway, E., "Delocalized plans and program comprehension", *IEEE Software*, vol. 19, no. 3, May 1986, pp. 41 - 48.

[91] Littman, D. C., Pinto, J., Letovsky, S., and Soloway, E., "Mental models and software maintenance", in *Empirical Studies of Programmers*, Soloway, E. and Iyengar, S., Eds., Norwood N.J. Albex Publisher Coop., 1986, pp. 80 - 98.

[92] Lloyd, P., Lawson, B., and Scott, P., "Can concurrent verbalization reveal design cognition?" *Design Studies*, vol. 16, 1995, pp. 237-159.

[93] Martin, R. C., *Agile Software Development, Principles, Patterns, and Practices*, Maassachusettes, Addison Wesley, 2002.

[94] Mateis, C., Stumptner, M., Wieland, D., and Wotawa, F., "Model-based debugging of Java programs", in Proceedings of 4th International Workshop of Automated and Algorithmic Debugging, 2000, pp. 32-40.

[95] Mayer, R., "The psychology of how novices learn computer programming", *ACM Computing Surveys*, vol. 13, no. 1, 1981, pp. 121-141.

[96] McDowell, C., Werner, L., Bullock, H., and Fernald, J., "The effects of pair-programming on performance in an introductory programming course", in Proceedings of SIGCSE Technical Symposium on Computer Science Education, Cincinnati, Kentucky, 2002, pp. 38-42.

[97] Mckeithen, K. B., Reitman, J. S., Reuter, H. H., and Hirtle, S. L., "Knowledge organization and skill differences in computer programmers", *Cognitive Psychology*, vol. 13, no. 307-325, 1981.

[98] Meijer, J. and Riemersma, F., "Analysis of solving problems", *Instructional Science*, vol. 15, 1986, pp. 3-19.

[99] Microsoft, "Microsoft Producer for Microsoft Office Powerpoint 2003", http://www.microsoft.com/windows/windowsmedia/technologies/producer.aspx,

[100] Mosemann, R. and Wiedenbeck, S., "Navigation and comprehension of programs by novice programmers", in Proceedings of the 9th International Workshop on Program Comprehension, Toronto, Canada, May 12 - 13 2001, pp. 79-88.

[101] Muller, M. and Tichy, W., "Case study: extreme programming in a university environment", in Proceedings of 23rd International Conference on Software Engineering, Toronto, Canada, May 2001, pp. 537-544.

[102] Nawrocki, J. and Wokciechowski, A., "Experimental evaluation of pair programming", in Proceedings of European Software Control and Metrics (Escom), London, England, April 2001, pp. 269-276.

[103] Newell, A. and Simon, H. A., *Human Problem Solving*, New York, Prentice-Hall, 1972.

[104] Nitko, A. J., *Educational Assessment of Students*, 2nd ed., Englewood Cliffs, Prentice-Hall, 1996.

[105] Nosek, J. T., "The case for collaborative programming", *Communications of the ACM*, vol. 41, no. 3, 1998, pp. 105-108.

[106] Novak, J. D., *Learning, Creating, and Using Knowledge*, Mahwah, NJ, Lawrence Erlbaum Associates, 1998.

[107] Olson, G. M., Olson, J. S., Carter, M. R., and Storrosten, M., "Small group design meetings: an analysis of collaboration", *Human Computer Interaction*, vol. 7, no. 2, 1992, pp. 347-374.

[108] Orne, M. T., "Demand characteristics and the concept of quasi-controls", in *Artifact in Behavioral Research*, New York, NY Academic Press, 1969, pp. 143-179.

[109] Ostwald, J., *Knowledge Construction in Software Development: The Evolving Artifact Approach*, University of Colorado, Ph.D. Dissertation, Boulder, USA, 1996.

[110] Parnas, D. L., "The limits of empirical studies of software engineering", in Proceedings of the International Symposium on Empirical Software Engineering, Nara, Japan, 2003, pp. 2-5.

[111] Pennington, N., "Comprehension strategies in programming", in *Empirical Studies of Programmers: Second Workshop,*, G. M. Olson, S. Sheppard, and Soloway, E., Eds., Norwood, NJ: Ablex. Publisher Coop., 1987, pp. 100-113.

[112] Perkins, D. N. and Martin, F., "Fragile knowledge and neglected strategies in novice programmers", in *Empirical Studies of Programmers*, Soloway, E. and Iyengar, S., Eds., Norwood N.J. Albex Publisher Coop., 1986, pp. 213-229.

[113] Peslak, A., "Teaching software engineering through collaborative methods", *Issues in Information Systems*, vol. 1, no. 1, 2004, pp. 247-253.

[114] Piaget, J., *The Construction of Reality in the Child*, New York, Basic Books, 1954.

[115] Rajlich, V., "Program comprehension as a learning process", in Proceedings of the First IEEE International Conference on Cognitive Informatics, Calgary, Canada, 2002, pp. 343-350.

[116] Rajlich, V. and Bennett, K. H., "A staged model for the software lifecycle", *Computer*, vol. 33, no. 7, July 2000, pp. 66-71.

[117] Rajlich, V., Doran, J., and Gudla, R., "Layered explanations of software: a methodology for program comprehension", in Proceedings of the Third International Workshop on Program Comprehension, Washington, D.C., 1994, pp. 46-52.

[118] Rajlich, V. and Gosavi, P., "Incremental change in Object-Oriented Programming", *IEEE Software*, vol. July/August, 2004, pp. 62-69.

[119] Rajlich, V. and Xu, S., "Analogy of incremental program development and constructivist learning", in Proceedings of the Second IEEE International Conference on Cognitive Informatics, London, UK, 2003, pp. 142-150.

[120] Rajlich, V. and Xu, S., "Constructivist learning during software development", *ACM TAAS Special Issue on CI and Autonomic Computing* no. Submitted, 2006.

[121] Ramalingam, V. and Wiedenbeck, S., "An Empirical Study of Novice Program Comprehension in the Imperative and Object-Oriented Styles", *Empirical Studies of Programmers, Ablex Publishing Corporation, Norwood,NJ* 1997, pp. 124-139.

[122] Robbin, J. E., *Cognitive Support Features for Software Development Tools*, Ph.D. dissertation, University of California, Irvine, USA, 1999.

[123] Robbins, J. E., Hilbert, D. M., and Redmiles, D. F., "Extending design environments to software architecture design", *Automated Software Engineering*, vol. 5, no. 3, 1998, pp. 261-290

[124] Robillard, P. N., "The role of knowledge in software development", *Communications of ACM*, vol. 42, no. 1, January 1999, pp. 87-92.

[125] Robillard, P. N., Detienne, F., d'Astous, P., and Visser, W., "Measuring cognitive activities in software engineering", in Proceedings of the 20th International Conference on Software Engineering, Kyoto, Japan, April 19-25 1998, pp. 292-300.

[126] Rosenthal, R., "The effects of early data returns on data subsequently obtained by outcome biased experimenter", *Sciometry*, vol. 26, no. 4, 1963, pp. 497-493.

[127] Rosenthal, R., *Experimenter Effects in Behavioral Research*, New York, NY, Appleton Century Crofts, 1966.

[128] Rugaber, S., "The use of domain knowledge in program understanding", *Annals of Software Engineering*, vol. 9, 2000, pp. 143-192.

[129] Rugaber, S., Ornburn, S. B., and LeBlanc, R. J., "Recognizing design decisions in programs", *IEEE Software*, vol. 7, no. 1, January 1990, pp. 46-54.

[130] Sayyad-Shirabad, J. and Lethbridge, T., "A little knowledge can go a long way towards program understanding", in Proceedings of Fifth International

Workshop on Program Comprehension, Dearborn, MI, May 28-30 1997, pp. 19-28.

[131] Schneider, J.-G. and Johnston, L., "eXtreme Programming at universities - an educational perspective ", in Proceedings of 25th International Conference on Software Engineering, Portland, Oregon, May 3-10 2003, pp. 594-599.

[132] Shaft, T. M. and Vessey, I., "The relevance of application domain knowledge: the case of computer program comprehension", *Information Systems Research*, vol. 6, 1995, pp. 286-299.

[133] Shepherd, A., "Hierarchical task analysis and training decisions", *Programmed Learning and Educational Technology*, vol. 22, 1985, pp. 162-176.

[134] Shneiderman, B., "Exploratory experiments in programmer behavior", *International Journal of Computer and Information Sciences*, vol. 5, no. 2, June 1976, pp. 123-143.

[135] Shneiderman, B., "Measuring computer program quality and comprehension", *International Journal of Man-Machine Studies*, vol. 9, 1977, pp. 465--478.

[136] Shneiderman, B., *Software Psychology*, Winthrop Publishers Inc., 1980.

[137] Shneiderman, B. and Mayer, R., "Syntactic/semantic interactions in programmer behaviour: a model and experimental results", *International Journal of Computer and Information Sciences*, vol. 8, no. 3, 1979, pp. 219 - 238.

[138] Sim, E. S. and Storey, M.-A. D., "A structured demonstration of program comprehension tools", in Proceedings of Seventh Working Conference on Reverse Engineering, Brisbane, Australia, November 23 - 25 2000, pp. 184-193.

[139] Singer, J., Elves, R., and Storey, M.-A., "NavTracks: supporting navigation in software", in Proceedings of 13th International Workshop on Program Comprehension, St. Louis, MO, USA, 15-16 May 2005, pp. 173-175.

[140] Soloway, E., "What to do next: meeting the challenge of programming-in-the-large", *Empirical Studies of Programmers, Ablex Publishing Corporation, Norwood,NJ* 1986, pp. 263-268.

[141] Soloway, E. and Ehrlich, K., "Empirical studies of programming knowledge", *IEEE Transactions on Software Engineering*, vol. 10, no. 5, September 1984, pp. 595-609.

[142] Soloway, E., Pinto, J., Letovsk, S., Littman, D. C., and Lampert, R., "Design documentation to compensate for delocalized plans", *Communication of the ACM*, vol. 31, no. 11, 1988, pp. 1259-1267.

[143] Sommerville, I., *Software Engineering*, Addison-Wesley, 1996.

[144] SouceForge.net, "JAdvisor: http://jadvisor.sourceforge.net/",

[145] Spohrer, J. G. and Soloway, E., "Analyzing the high frequency bugs in novice programs", in *Empirical Studies of Programmers*, Soloway, E. and Iyengar, S., Eds., Norwood N.J. Albex Publisher Coop., 1986, pp. 230-251.

[146] Srikanth, H., Williams, J. G., Wiebe, E., Miller, C., and Balik, S., "On pair rotation in the computer science course", in Proceedings of the 17th

Conference on Software Engineering Education and Training, Norfolk, Virginia March 1-3 2004, pp. 144-149.

[147] Storey, M.-A. D., Fracchia, F. D., and Müller, H. A., "Cognitive design elements to support the construction of a mental model during software exploration", *The Journal of Systems and Software*, vol. 44, no. 3, 1999, pp. 171--185.

[148] Stotts, P. D., Williams, L. A., Nagappan, N., Baheti, P., Jen, D., and Jackson, A., "Virtual teaming: experiments and experiences with distributed pair programming", in Proceedings of Third XP and Second Agile Universe Conference, New Orleans, LA, August 10-13 2003, pp. 129-141.

[149] Tiene, D. and Ingram, A., *Exploring Current Issue in Educational Technology*, New York, NY, McGraw-Hill, 2001.

[150] Tuck, D., France, R., and Rumpe, B., "Limitations of agile software processes", in Proceedings of the Third International Conference on Extreme Programming and Flexible Processes in Software Engineering (XP2002), Alghero, Italy, 2002, pp. 43-46.

[151] Uchida, S., Monden, A., Iida, H., Matsumoto, K., Inoue, K. and Kudo, H, "Debugging process models based on changes in impressions of software modules", in Proceedings of International Symposium on Future Software Technology, Guiyang, China, August 2000, pp. 57-62.

[152] Vans, A., von Mayrhauser, A., and Somlo, G., "Program understanding behavior during corrective maintenance of large-scale software",

International Journal of Human-Computer Studies, vol. 51, no. 1, July 1999, pp. 31-70.

[153] Vessey, I., "Expertise in debugging computer programs", *International Journal of Man-Machine Studies*, vol. 18, 1983, pp. 459-494.

[154] Visser, W., "Strategies in programming programmable controllers: a field study on professional programmer", *G.M.Olson, S.Sheppard and E.Soloway (eds), Empirical studies of programmers: second workshop, Ablex Publishing Corporation, Norwood,NJ* 1987, pp. 217-230.

[155] von Glasersfeld, E., *Radical Constructivism*, London, England, The Falmer Press, 1995.

[156] Von Mayrhauser, A. and Lang, S., "A coding scheme to support analysis of software comprehension", *IEEE Transactions on Software Engineering*, vol. 25, no. 4, 1999, pp. 526-540.

[157] von Mayrhauser, A. and Vans, A. M., "Comprehension processes during large scale maintenance", in Proceedings of International Conference of Software Engineering, Sorrento, Italy, May 1994, pp. 39-48.

[158] Von Mayrhauser, A. and Vans, A. M., "Identification of dynamic comprehension processes during large scale maintenance", *IEEE Transactions on Software Engineering*, vol. 22, no. 6, 1996, pp. 424-437.

[159] Von Mayrhauser, A. and Vans, A. M., "Program understanding behavior during debugging of large scale software", in Proceedings of Seventh Workshop on Empirical Studies of Programmers, Alexandria, VA, Oct 24- 26 1997, pp. 157–179.

[160] Von Mayrhauser, A. and Vans, A. M., "Program understanding behavior during adaptation of large scale software", in Proceedings of the Sixth International Workshop on Program Comprehension, Los Alamitos, CA, 1998, pp. 164-172.

[161] Vygotsky, L. S., *Mind in Society*, Cambridge, MA, Harvard University Press, 1978.

[162] Wallenstein, A., *Cognitive Support in Software Engineering Tools: A Distributed Cognition Framework*, Ph.D. Dissertation, Simon Fraser University, 2002.

[163] Wallenstein, A., "Observing and measuring cognitive support: steps toward systematic tool evaluation and engineering", in Proceedings of the 11th International Workshop on Program Comprehension, Portland, Oregon, 2003, pp. 185-195

[164] Wang, Y., "On cognitive informatics", in Proceedings of the First IEEE International Conference on Cognitive Informatics, Calgary, AB, 2002, pp. 34-42.

[165] Wang, Y., "On the cognitive informatics foundations of software engineering", in Proceedings of the 3rd IEEE International Conference on Cognitive Informatics, Victoria, BC, 2004, pp. 22-31.

[166] Watson, J. B., "Is thinking merely the action of language mechanism?" *British Journal of Psychology*, vol. 11, 1920, pp. 87-104.

[167] Wiedenbeck, S. and Ramalingam, V., "Novice comprehension of small programs written in the procedural and object-oriented styles", *International Journal of Human-Computer Studies*, vol. 51, no. 1, July 1999, pp. 71-87.

[168] Wilde, N. and Scully, M., "Software reconnaissance: mapping program features to code", *Software Maintenance: Research and Practice*, vol. 7, 1995, pp. 49-62.

[169] Williams, J. G. and UpChirch, R. L., "In support of student pair-programming", in Proceedings of the 32th SIGCSE Technical Symposium on Computer Science Education, Charlotte, NC, February 21-25 2001, pp. 327 - 331.

[170] Williams, L., Kessler, R., Cuningham, W., and Jeffries, R., "Strengthening the case for pair-programming", *IEEE Software* July/August 2000, pp. 19-25.

[171] Williams, L., McDowell, C., Nagappan, N., Fernald, J., and Werner, L., "Building pair programming knowledge through a family of experiments", in Proceedings of International Symposium on Empirical Software Engineering, Rome, Italy, September 29-October 4 2003, pp. 143-152.

[172] Xu, S., "A cognitive model for program comprehension", in Proceedings of the 3rd ACIS International Conference on Software Engineering, Research, Management & Applications (SERA2005), Mount. Pleasant, Michigan, August 11-13 2005, pp. 392-398.

[173] Xu, S. and Rajlich, V., "Cognitive process during program debugging", in Proceedings of the Third IEEE International Conference on Cognitive Informatics, Victoria, BC, August 16-17 2004, pp. 176-182.

[174] Xu, S. and Rajlich, V., "Dialog-based protocol: an empirical research method for cognitive activity in software engineering", in Proceedings of the 4th ACM/IEEE International Symposium on Empirical Software Engineering, Noosa Heads, Queensland, November 17-18 2005.

[175] Xu, S. and Rajlich, V., "Pair Programming in Graduate Software Engineering Course Projects", in Proceedings of 2005 ASEE/IEEE Frontiers in Education Conference (FIE 2005), Indianapolis, Indiana, October 19-22 2005, pp. p.FIG-7-FIG-12.

[176] Xu, S. and Rajlich, V., "Programmer learning during software development: empirical study using dialog-based protocols and self-directed learning theory", *IEEE Transactions on Software Engineering*, vol. submitted, 2006.

[177] Xu, S., Rajlich, V., and Marcus, A., "An empirical study of programmer learning during incremental software development", in Proceedings of the 4th IEEE International Conference on Cognitive Informatics, Irvine, California, August 8-10 2005, pp. 340-349.

[178] Yin, R. K., *Case Study Research: Design and Methods*, Thousand Oaks, CA, Sage Publications Inc., 1994.

[179] Yoon, B. and Garcia, O. N., "Cognitive activities and support in debugging", in Proceedings of 4th Annual Symposia on Human Interaction with Complex System, Los Alamitos, CA, 1998, pp. 160-169.

[180] Zayour, I., *Reverse Engineering: A Cognitive Approach, A Case Study and a Tool, Ph.D. Dissertation*, University of Ottawa, Ottawa, Canada, 2002.

ABSTRACT

COGNITIVE ASPECTS OF SOFTWARE ENGINEERING PROCESSES

by

SHAOCHUN XU

August 2006

Advisor: Dr. Vaclav Rajlich

Major: Computer Science

Degree: Doctor of Philosophy

Software engineering activities are to process a large amount of knowledge and therefore, the cognitive process is mainly involved. Studying the cognitive process involved in software engineering can greatly help us to understand the whole process and to improve software engineering research.

In order to study the cognitive process, we developed an empirical method that includes dialog-based protocol and self-directed learning theory. The dialog-based protocol is based on the analysis of the dialog that occurs between programmers in pair programming. It is an alternative to the common think-aloud protocol and it may minimize the Hawthorne and placebo effects. The self-directed learning theory is based on the constructivist learning theory and the Bloom taxonomy. It captures the specifics of the programmer's cognitive activities and provides an encoding scheme used in analyzing the empirical data.

We conducted a case study of expert and intermediate programmers during incremental software development. Compared to intermediate

programmers, experts discussed more domain concepts at a greater length before starting to write code. Experts mostly concentrated on one concept at one time, while intermediate programmers often discussed several concepts simultaneously. Experts were willing to reconsider and correct obsolete design decisions, while intermediate programmers retained all design decisions. Experts had more absorption activities at higher Bloom levels such as analysis and synthesis, while intermediate programmers had more absorption activities at lower Bloom levels such as comprehension and application. Experts spent more time analyzing the knowledge and generating test cases, while intermediate programmers spent more time learning the knowledge.

We also conducted a case study on program debugging, and found that programmers apply the cognitive activities at all six Bloom levels and move from lower one to upper one in order to make update. Program debugging is a more complex activity than incremental software development.

A case study on pair programming in software evolution class projects was also performed. The results of the case study showed that paired students completed their change request tasks faster and with higher quality than individuals. They also wrote less lines of code and used more meaningful variable names.

AUTOBIOGRAPHICAL STATEMENT

SHAOCHUN XU

Shaochun Xu is a Ph.D. candidate in the Department of Computer Science at Wayne State University in Detroit. He was born in Hubei Province, China. He received his Bachelor of Science degree from Department of Geology, Peking University, China in 1984 and Master of Science degree in 1987. He got a Ph.D. in Geology from Liege University, Liege, Belgium in 1997. After he completed all the undergraduate requirements in Bachelor of Computer Science in the University of Manitoba, Canada, he was admitted into Master of Science in Computer Science program in the University of Windsor, Canada. He got his Master degree in Computer Science in 2001.

He is now an assistant professor in Algoma University College, Laurentian University, Sault Ste. Marie, Canada. He teaches Data Structures, Software Engineering, Introduction to Computer Science and Introduction to Database Programming courses.

Before coming to Wayne State, he has conducted a few years postdoctoral research in the Department of Geological Science of the University of Manitoba, and has been an instructor in the School of Computer Science in University of Windsor.

His current research interests include software maintenance and cognitive process in software engineering.