

# Dialog-Based Protocol: An Empirical Research Method for Cognitive Activities in Software Engineering

Shaochun Xu, Václav Rajlich  
Department of Computer Science  
Wayne State University  
Detroit, Michigan USA 48202  
{shaochun, rajlich}@wayne.edu

## Abstract

*This paper proposes dialog-based protocol for the study of the cognitive activities during software development and evolution. The dialog-based protocol, derived from the idea of pair programming, is a significant alternative to the common think-aloud protocol, because it lessens the Hawthorne and placebo effects. Using screen-capturing and voice recording instead of videotaping further reduces the Hawthorne effect. The self-directed learning theory provides an encoding scheme and can be used in analyzing the data. A case study illustrates this new approach.*

## 1. Introduction

Software is a human-intensive technology and the studies of cognitive processes in software engineering can shed light on many software engineering problems [30]. A common characteristic of software engineering processes is the handling of large amounts of knowledge distributed over several domains, such as the problem domain and programming techniques domain [9]. Cognitive processes that handle this knowledge become a critical issue of a practical interest.

Empirical studies have been widely applied in software engineering research [2]. Some of these issues of great importance are how to conduct the experiments, how to capture the data, and how to analyze the data [19]. The empirical approaches, which have been commonly used by cognitive scientists, include protocol analysis, task analysis, and discourse analysis [13, 14, 34].

Task analysis is one of the techniques that study both the subject's actions and the accompanying cognitive processes [34]. It is commonly used with users who have problems mastering complex behaviors and 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 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 [14]. It is neither a qualitative nor a quantitative research method, but a manner of questioning the basic assumptions. 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 [13]. It studies human cognitive processes and identifies basic knowledge objects. Think-aloud protocol analysis 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 [36]. However, there are some deficiencies in this approach, such as placebo effect [32] and Hawthorne effect [1]. For that reason, the researchers have to search for additional empirical methods that may offer a relief from these problems.

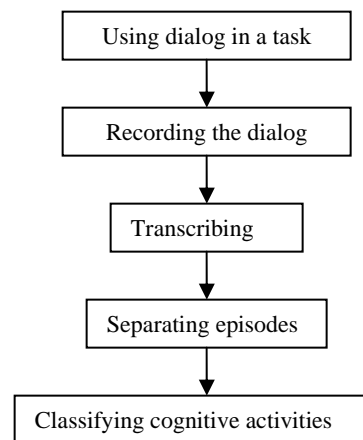


Figure 1: Steps of dialog-based protocol

In this paper, we present a new empirical method, dialog-based protocol analysis, see Figure 1. The idea of dialog-based protocol comes from the pair programming

that is one of the practices of Extreme Programming [3]. Voice recording captures the pair dialog, screen-capturing software captures the sequence of computer screens, and the self-directed learning theory [39] provides an encoding scheme and is used in analyzing the data.

Section 2 of this paper describes the think-aloud protocol and contrasts it with our dialog-based protocol, while Section 3 explains the recording methods. Section 4 discusses the data transcribing and processing. Section 5 contains the case study. Section 6 surveys the strengths and weaknesses of the new approach. The conclusions and future work are in Section 7.

## 2. Think-Aloud and Dialog-Based Protocols

In protocol analysis, subjects give verbal reports on the cognitive processes that they are experiencing during a task. The common three types of protocol reports are introspective, retrospective, and think-aloud [17]. Introspective and think-aloud require subjects to report their thoughts during a task, whereas retrospective protocol reports are gathered after a task is finished. Both introspective and retrospective reports require the subjects to interpret the cognitive processes they use [13]. However in think-aloud setting, 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.

These three types of verbal protocols refer to one subject working individually. They can provide insights that cannot be obtained by other methods such as observation and interviewing [17].

### 2.1. Think-Aloud Protocol

The think-aloud protocol was documented in 1920's when Watson used it to illustrate general characteristics of cognitive process in problem solving [37]. The approach was later developed into a research method, especially when tape recorders became available in 1945. For example, Duncker (1945) used it to analyze problem-solving processes in terms of memory search [12]. The think-aloud method was systematically described by Newell and Simon (1972) and used in combination with computer models to build a detailed model of problem-solving processes [23].

Think-aloud protocol assumes that the subjects can verbalize their thoughts without changing the sequence of their tasks; therefore, the recorded verbalization can be accepted as valid data. Ericsson and Simon (1993) concluded that the closest connection between thinking and verbal reports is found when subjects verbalize thoughts generated during a task completion [13]. They also mentioned that there is no evidence showing that the

sequences of thoughts had been changed when subjects thought aloud comparing with subjects who completed the same tasks silently [13].

Think-aloud method provides a way to collect data which could hardly be collected by other means. It not only allows us to understand what the subjects are doing, but also why. It provides a basis for investigating the underlying mental process of complex tasks.

Although think-aloud protocol was originally used in psychology [13], it has now been used in other areas, such as medicine, usability evaluation [4], engineering design [16] and software comprehension [35]. In most cases, think-aloud questions are used to stimulate verbalization, such as "what are you thinking now", "keep talking" "think aloud" etc.

Think-aloud protocol forces the subjects to speak what they have in mind. Although Ericsson and Simon (1993) did not find changes in the cognitive process when subjects were asked to verbalize their thoughts, they found that certain changes occurred when subjects were asked to explain their cognitive process [13].

A serious problem with the think-aloud is that the subjects may endeavour to deliberately produce the desired data [32]. This effect is called "placebo effect". The outcome of an experiment may be biased if the experimenter unintentionally indicates the expectations by verbal communications or gestures, for example nodding when an expected result is observed [31].

Another problem is caused by the presence of experimenter/mentor which might also affect the behavior of the subjects. Orne (1969) stated that subjects can never be neutral to an experiment [25]; the Hawthorne effect [1] occurs when the subjects know that they are being studied. Think-aloud protocol with observation and the communication between mentor and subjects might particularly affect the outcome [32]. A verbal report of the mental process may change the way how a subject interacts with the task and affect the subject's decision process. Parnas [26] discussed the limitations of empirical studies in software engineering due to the Hawthorne effect and other factors.

### 2.2. Dialog-Based Protocol

In order to reduce the defects of think-aloud protocol, we propose a dialog-based protocol. Dialog is a conversation between two people, which allows them to share ideas and to learn from each other. Dialog contains a large quantity of information which has been used as the raw data for studies in many fields, such as linguistic, psychology, sociology, and literature [8].

During pair programming, two programmers work side-by-side on design, algorithm selection, implementation, and testing at one machine. Since both programmers

cooperate with each other, they need to verbalize their thoughts in order to enable their partner to understand.

We can record their dialog and their activities, and analyze the recording in order to study the cognitive activities. We call this method “dialog-based protocol”. It is derived from the idea of pair programming [3] and it is in many ways similar to think-aloud protocol. One major difference between the two protocol techniques lies in the fact that in the think-aloud protocol, only one subject is required to speak, whereas in the dialog-based protocol, two people are verbally expressing their thoughts while completing the tasks. Both protocols require the process to be recorded, either as audio or video, for analysis.

In order to evaluate the usability of this protocol, we discuss some anticipated advantages.

### 2.3. Anticipated Comparisons

According to Ericsson and Simon (1993), think-aloud protocol can only produce a subset of thoughts [13]. We speculate that dialog-based protocol can force both partners to explain all essential issues clearly to their partners and allow a more complete process to be documented. Also, dialog-based protocol requires continuous verbalization during programming task, which could provide a more complete picture of cognitive process than think-aloud protocol because the latter is very likely to be constantly disrupted by the experimenter. However, we realize that even in dialog-based setting the recorded dialogs might not be complete as some unconscious thought might be still missing from the dialog.

Think-aloud protocol forces subjects to speak and interferes with the thinking process [20]. Some subjects may experience difficulties talking when they perform their tasks, or be affected by the presence of the mentor [13]. Some people may find it difficult to verbalize their every thought [4]. We speculate that dialogs can be recorded under more natural circumstances than a think-aloud sessions. When people collaborate, they are forced to give arguments in order to clarify their thinking, and therefore they can easily forget that they are recorded, and that should reduce the Hawthorne effect.

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 process. In dialog-based setting, there is no communication between mentor and subjects. Therefore, we speculate that the invalid portion of data is reduced.

Another problem associated with think-aloud setting of data is the misinterpretation. Some subjects may have difficulties in explaining the process. However, in

dialog-based session, the subjects have to explain all important points clearly to their partners and as a result, such misinterpretations could be reduced.

**Table 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	affected	unaffected
Mentor Presence	necessary	unnecessary
Pairing problem	does not exist	exists

The think-aloud protocol may also impact the programmers’ performance, because they have to conduct the tasks and also verbalize them. Berry and Broadbent reported that people who were not thinking aloud performed faster than those who were [5]. In dialog-based protocol, the programming pair conducts the task and talks at the same time. There is no additional and artificial overhead that would impact productivity.

The mentor must be present in think-aloud protocol since he/she might need to remind the subjects to verbalize the thinking. For the dialog-based protocol, the presence of mentor is not necessary since both programmers need to communicate anyway. Therefore, the cost of the presence of the mentor during the experiment is higher in think-aloud protocol, than in dialog-based protocol.

Pairing is an issue in dialog-based protocols. 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.

Table 1 shows the anticipated comparisons between think-aloud protocol and dialog-based protocol.

## 3. Recording Methods

The recording methods might affect the experiment, particularly when the subjects know that they are being recorded and might alter their behavior. Therefore, choosing the right recording method for data collecting is an important issue.

Among the recording methods, video and audio recordings are most commonly used. They provide rich and relatively permanent primary records.

### 3.1. Videotaping

Videotaping produces a continuous view of events and provides a detailed and unique record of the process. 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 [13]. Videotaping can record detailed movement of the mouse, specific events, the code changes and of course, the communication between programmers. When analyzing video data, individual scenes can be replayed numerous times as the researcher reflects on what has occurred, thus reducing the possibility of premature conclusions [13].

Video provides continuous data rather than snapshot data. This is of great benefit because it records the richness of interactions. 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 several limitations and weaknesses that need consideration.

Subjects may be more awkward or shy in front of a video camera than if 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 experimenter than playback of digital audio. Digital camcorders require a learning curve to operate successfully. A lot of work is involved when video data is transferred to a computer, such as 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, learning curve, and 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. Due to the huge amount of information being processed, video clips processing requires higher level computer, while a low-end PC would perform the task satisfactorily in audio production.

### 3.2. Voice-Recording

Under certain circumstances, voice-recording is preferred to videotaping, due to the limitation of videotaping mentioned in the previous section. The advantages of voice-recording method are commonly accepted as:

- Easy to conduct
- Easy to play back
- Minimal Hawthorne effect since the microphone can be put in a hidden place
- Less costly

Of course, voice-recording cannot record the mouse movements and the screen changes made by programmers.

### 3.3. Software Screen-Capturing

For 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. Therefore the screen-capturing software evolved into an effective tool for collecting data during cognitive research.

Screen-capturing does not require any extensive operation knowledge since it creates media files which can be operated easily. On the other hand, there can be a few seconds' delay in the screen change capture, therefore, there is a possibility that some fast changes happened during this time and are not found in the screenshots. However, since programmers can not write a line of code (or make change) during seconds, such delay does not give any substantial negative effects when transcribing the data.

An example of screen-capturing software is Microsoft Producer [22], 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 essential data during programming process. Microsoft Producer delivers high quality audio and video files in various formats. In this way, the programming process with its environment can be stored or replayed for further analysis.

## 4. Transcription and Analysis

After the data is captured, it is transcribed into "raw protocol". Similarly as in think-aloud setting, the raw protocol is then divided into small units called "segments" or "episodes".

### 4.1. Producing Episodes

Raw protocols are often divided into segments according to verbalization events or syntactic markers [13]. Some researchers segmented the raw protocols based on the subject's intentions and actions [16].

In our case, we propose the dialog to be divided into episodes based on concepts that the dialog deals with. We classify the concepts as domain concepts and programming concepts. Domain concepts belong to a specific domain which the program addresses [6], such as *strike* in the bowling application, while programming concepts belong to the knowledge of programming, such as the programming language, program development process, design decisions, and so forth. We further propose to consider only design decisions as programming concepts since design decisions are the concepts that drive the program development, while the other programming concepts only play auxiliary role.

We use the following rule to separate dialog into episodes: a new episode begins when programmers start 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 part of the dialog, we combine this part with the previous episode.

#### 4.2. Protocol Analysis with a Coding Scheme

After the data is transcribed and divided into episodes, data analysis can be conducted.

Some researchers did not use a coding scheme to analyze data in think-aloud protocol. For example, Meijer and Riemersma (1986) and Confrey (1994) analyzed think-aloud protocols and supported their conclusions by directly citing the protocols [21] [10]. 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 [15].

In software engineering, von Mayrhauser and Lang (1999) pointed out that one of the difficulties for protocol analysis is to compare the results across different studies [35]. That is why a coding scheme is needed to provide a standard classification. The scheme should be carefully chosen so as to allow the replication of the protocol and the assessment of the generality of its finding.

Pennington (1987) has proposed a coding scheme for program comprehension analysis [27]. Oslo et al (1992) developed a coding scheme to observe the programmer behavior during design meeting [24]. Shaft (1995) developed a coding scheme or “coder’s manual” for her experiment in order to assess the process of coding the verbal data [33]. Most recently, von Mayrhauser and Lang (1999) described a detailed coding scheme, AFECS, which uses explicit codes [35]. In their scheme, the codes are split into segments, which provide flexibility, expandability, and ease of automation.

Rajlich and Xu [39] proposed a self-directed learning theory which can be used as a formal coding scheme. The self-directed learning combines the constructivist learning theory [28] and Bloom’s taxonomy of the cognitive

domain [7]. 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 2: 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

Constructivist learning is based on the work of Piaget on child learning [28]. Piaget classified two learning activities: assimilation and accommodation. Assimilation refers to the way learners deal with new knowledge and accommodation describes how learners reorganize their existing knowledge. Rajlich and Xu [29] divided assimilation into two separate activities: absorption, which refers to the case when learners add new facts to their knowledge, and denial, which occurs when learners reject the new facts because they do not fit in with the existing knowledge (see Table 2). Accommodation was also divided into two separate activities. When the learners reorganize their knowledge to aid future absorption of new facts, we call their cognitive activity reorganization. It is called expulsion when a part of the knowledge becomes obsolete or provably incorrect and the learners reject it.

Bloom and his colleagues identified six levels of learning known as the Bloom’s taxonomy of the cognitive domain; see Table 3 [7]. The six levels include the knowledge or recognition of facts at its lowest level, then comprehension, application, analysis, synthesis, and evaluation [7]. It is a well-established part of the theory of learning and we applied it to classify the levels of the programmer cognitive activities.

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’s taxonomy.

Once the dialog has been decomposed into episodes, the self-directed learning theory is used for protocol analysis. The first part of analysis involves assignment of activity types as specified by the self-directed learning theory. Each episode is classified according to the four cognitive activities (i.e., absorption, denial,

reorganization, and expulsion) by using the characteristic verbs of Table 2. Some episodes might be assigned more than one cognitive activity.

**Table 3: The Six Bloom’s Levels and Corresponding Verbs**

Bloom’s levels	Sample verbs
Recognition	collect, copy, define, describe, enumerate, examine, identify, label, list, name, quote, read, recall, retell, record, repeat, reproduce, select, state, tell
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
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
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
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
Evaluation	appraise, assess, conclude, criticize, convince, decide, defend, discriminate, evaluate, explain, grade, judge, justify, measure, rank, recommend, reframe, support, test, validate, verify

Then, the Bloom’s levels are assigned to each episode also through the use of characteristic verbs of Table 3. Table 3 is a composite of similar tables in [11, 18] and it was cleaned for the purposes of software engineering, where some verbs attained a different meaning, for example verb *compile*.

Given the qualitative nature of the coding process, two or more coders are proposed to do the coding assignments because disagreements in the classifications are possible. Then the results are compared and the disagreements are highlighted and discussed in order to achieve a consensus.

The final data is computed as the frequencies of occurrences in episodes for each activity type and for each Bloom’s level.

## 5. Case Study

In order to illustrate the new empirical research method, we present a brief overview of a case study in incremental software development. In the case study, we asked ten programmer pairs to implement an application

using pair programming, test-first driven and refactoring techniques. We used videotaping for two pairs and software screen-capturing for eight pairs. Due to space limitation, here we only briefly summarize the case study results. We have reported the results for the early two videotaped pairs in more detail in [39]. We will discuss the observed strengths and weaknesses of the dialog-based protocol and the recording methods in Section 6.

### 5.1. Participants

Sixteen participants were graduate students from the Department of Computer Science at Wayne State University, who were taking a graduate software engineering course (i.e., CSC7110 - Software Engineering Environments), and four participants were advanced undergraduate students who were taking a software engineering course (CSC4110 - Software Engineering). All participants were classified as intermediate level programmers. They voluntarily joined the case study as a part of their course projects. They had intermediate-to-large programming experience with C, C++, and Java, but they never used pair programming or test-first and refactoring practices.

The data from two pairs were rejected as invalid, as discussed in Section 6 in more detail. The results of the remaining eight pairs were compared to an earlier published study of expert programmers who solved the same task [39].

### 5.2. Material

The problem to be solved in the case study is to implement an application which records the bowling scores for the bowling game. It requires the programmer pair to understand both the relevant domain concepts and program concepts.

**Table 4. Main characteristics of the program and dialog of one of the pairs**

Items	data
Lines of code	222
Number of class members	25
Number of classes	3
Number of domain concepts discussed before coding started	3
Number of concepts in the knowledge at the end of the task	32
Number of refactorings	9
Number of episodes	62

### 5.3. Procedure

The case study was carried over two semesters: two early pairs were videotaped in March 2004, and the rest used screen-capturing in February and March 2005. Since all programmers were new to pair programming, test-first programming and refactoring, a short training session was

**Table 5. The distribution of cognitive activities and Bloom's levels for the same pair**

	Assimilation		Accommodation		Total
	Absorption	Denial	Reorganization	Expulsion	
Recognition	0 (0.00%)	0 (0.00%)	0 (0.00%)	0 (0.00%)	0 (0.00%)
Comprehension	12 (19.35%)	1 (1.61%)	0 (0.00%)	0 (0.00%)	13 (20.96%)
Application	15 (24.20%)	0 (0.00%)	7 (11.29%)	0 (0.00%)	22 (35.49%)
Analysis	13 (20.97%)	0 (0.00%)	2 (3.23%)	0 (0.00%)	15 (24.20%)
Synthesis	12 (19.35%)	0 (0.00%)	0 (0.00%)	0 (0.00%)	12 (19.35%)
Evaluation	0 (0.00%)	0 (0.00%)	0 (0.00%)	0 (0.00%)	0 (0.00%)
Total	52 (83.87%)	1 (1.61%)	9 (14.52%)	0 (0.00%)	62 (100.00%)

given prior to the case study. During the training session, the programmers were provided with reading materials on pair programming, test-first, and refactoring techniques. They were asked to write a simple program in Eclipse platform with JUnit in order to be familiar with those tools.

One of the authors acted as the mentor and monitored the process, recorded the case study, and the programmers were provided with a description of the bowling rules.

After the case study was done, the authors transcribed the files into raw protocols, divided the dialogs into episodes, and classified the episodes according to the coding scheme.

#### 5.4. Results

It took the pairs from three 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. All pairs started the dialog with a discussion of two to six domain concepts before coding started, which reflected their intermediate programming ability. In comparison, expert programmers discussed more domain concepts on the same problem before the code was written [39].

Our pairs made from 24 to 47 design decisions during the process, and almost all of them were retained until the end. Since our programmers were new to the refactoring techniques, most of them only applied it few times at the beginning of the process by simply renaming the variables. One pair even did not use refactoring at all. Compared to that, experts often 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 [39].

Unlike experts who switched often from one concept to another during the dialog, our pairs discussed each concept for a longer time and this resulted in much fewer episodes in the dialogs [39].

As an illustration of the data collected, Table 4 shows the characteristics of the program implemented by one of the pairs. The pair was formed by two male graduate students who have mastered the case study requirements and communicated effectively during the process. They

often switched the roles, which also happens in expert programmer teams. The distribution of cognitive activities and Bloom's levels throughout of the recorded episodes of the same pair are presented in Table 5.

In the case study, the absorption activity dominates the programming process from the beginning to the end. For our pairs, absorption ranged from 83% to 98%, while there were only 68% absorption activities in the experts' work [39]. We speculate that our intermediate programmers spent much more effort to absorb new knowledge than the experts did. It is worthy to notice that most absorption activities of our pairs occur at the lower Bloom's levels (comprehension and application) in majority pairs.

Four out of six Bloom's levels have been found in our case study. More episodes were classified at the lower Bloom's levels (recognition, comprehension, and application) in the dialogs of our pairs than that of experts [39]. We speculate that intermediate programmers spent more time in understanding and applying the knowledge, while experts took part of the knowledge as granted and spent more time on analyzing and synthesizing the knowledge.

No episodes were classified at evaluation or at recognition levels of Bloom's taxonomy. However, Xu and Rajlich [38] previously identified all six levels of Bloom's taxonomy in a case study of program debugging.

#### 5.5. External Validity

Since we have repeated our case study with ten pairs and the results for each pair are similar, we are convinced that our case study can be easily replicated. However, our case study still has some limitations:

- The program problem solved by the programmers is relatively simple. For more complex problems, results may be different.
- Our pairs did not have much experience with refactoring and test-first practices, which may affect the outcome.
- The episodes were separated and classified manually by one of the authors, based on concepts. A different separation criterion (for example, based on activities rather than concepts) might yield different results.

- Different authors may separate and classify episodes differently.

## 6. Observed Strengths and Weaknesses of Our Approach

This section summarized the practical experience with our approach and compares it with the anticipations listed in Section 2.3, 3, and 4.

### 6.1. Dialog-Based Protocol

According to our observation, when two programmers in pair were well matched, they worked efficiently and conducted a revealing dialog. In our case study, we run into several difficulties which may 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 member wrote the program alone and explained his thoughts to the other member. His partner simply said “yes” or “right” all the time, almost 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.

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

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

We often observed cases where subjects did not understand their partners’ idea and the partners tried to illustrate and explain it, which gave a more detailed explanation about their thoughts. Whenever an ambiguity or misunderstanding arose, the 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 than in think-aloud protocol, confirming our earlier anticipation.

Since the mentor was simply an observer, the communication between mentor and subjects had been greatly reduced. Even when subjects had a technical question, they did not have to ask the mentor for help since their partners was always able to solve the problem. During the case study, the mentor was only asked questions at the beginning of the process when several subjects had problems with the use of the compiler.

As mentioned above, the mentor was only consulted by several pairs at the very beginning of the dialogs. There was no communication between the mentor and three of

the pairs. Therefore we believe it is not necessary to have mentor present for the dialog-based protocol, once subjects completely know program requirements and are familiar with the tools. We concluded that the placebo effect was greatly reduced.

In general, we concluded that dialog-based protocol is an effective method for collecting information about the programmers’ cognitive activities during software engineering processes, although there are still some open issues, such as how to form the pairs, and how to process the data fast. Based on our experience, we concluded that dialog-based protocol can be used to study the cognitive activities in software development. Our observations during the case study directly supported the anticipated comparisons between think-aloud and dialog-based protocols shown in Table 1.

### 6.2. Software Screen-Capturing

In our case study used videotaping for the first two pairs and then we switched to software screen-capturing for other eight pairs.

Videotaping provided more data, but these additional data did not play any significant role in our case study. We found several disadvantages of videotaping. Installation of a video-camera in a room increased the cost and decreased flexibility; we had to borrow and return the digital camera each time when it was used. The camera was prone to technical problems; once we had to delay the recording until the camera was repaired. Software screen-capturing solved those problems.

Recording quality with video camera turned out to be lower than using software screen-capturing. The camera had to be located in a distance from the subjects in order to capture whole the computer screen, but that increased the external noise in the recordings. Some screen data was 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 Hawthorne effect. We found no evidence of similar awareness during the 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 tape 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 difficulty to transcribe the media files into dialogs.

### 6.3. Coding 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 for studying the cognitive activities during software engineering processes.

Coding scheme uses limited selection of verbs for the four cognitive activities and six Bloom's learning levels (Tables 2 and 3); especially the verbs for Bloom's taxonomy have been well defined and used for many years.

Using self-directed learning theory, we were able to discover the differences between expert and intermediate programmers in terms of cognitive activities [39].

Similar to other coding schemes, however, there is a potential for disagreements. The verbs for the Bloom's levels have been developed and validated through many years of use, but still some verbs appear at more than one Bloom's levels, creating an ambiguous classification process. The verbs in Table 2 were used in our case study for the first time and might be not complete. Some episodes in recorded dialogs did not contain those verbs or even their synonyms, therefore not only the verb in the episode, but also the actual contents of the episode was considered during the classification.

## 7. Conclusions and Future Work

Developing and evolving software is a knowledge-intensive process. In response, there is a growing interest in the study of the cognitive activities during software engineering process. Empirical studies based on think-aloud protocol have been used for many years for that purpose. However, the think-aloud protocol has several drawbacks, especially the Hawthorne and placebo effects.

This paper proposes a new empirical method that consists of dialog-based protocol, software screen-capturing, and a coding scheme based on self-directed learning. The dialog-based protocol can greatly benefit software engineering empirical research by collecting more detailed and complete data, and reducing Hawthorne and placebo effects. The self-directed learning theory provides a detailed coding scheme for data analysis.

We documented some early experience with the dialog-based protocol, but many aspects of this methodology are still a matter of conjecture and more experience needs to be gained. In particular, we still need to investigate ways

how to form the pairs in order to facilitate the dialog, and refine and complete the verbs used in encoding.

In future, we would like to use our approach to conduct additional experiments on program debugging, program comprehension, and other software engineering processes where pair programming is applicable.

## 8. Acknowledgments

This work was partially supported by grant CCF-0438970 from National Science Foundation (NSF) and by IBM Eclipse Innovation Award 2005. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF or IBM.

## 9. References

- [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] Basili, V. R., "The role of experimentation in software engineering: past, present, future", in Proceedings of 18th International Conference on Software Engineering, Los Alamitos, CA, 1996, pp. 442-449.
- [3] Beck, K., *Extreme Programming Explained*, Massachusetts, Addison-Wesley, 2000.
- [4] Benbunan-Fich, R., "Using protocol analysis to evaluate the usability of a commercial web site", *Information & Management*, vol. 39, no. 2, 2001, pp. 151-163.
- [5] Berry, D. C. and Boardent, D. E., "The role of instruction and verbalization in improving performance on complex search tasks", *Behavior and Information Technology*, vol. 9, no. 3, 1990, pp. 175-190.
- [6] Biggerstaff, T. J., Mitbender, B. G., and Webster, D. E., "Program understanding and the concept assignment problem", *CACM*, vol. 37, no. 5, May 1994, pp. 72-82.
- [7] Bloom, B. S., "Taxonomy of Educational Objectives: The Classification of Educational Goals: Handbook, I, Cognitive Domain": New York, Toronto, Longmans, Green, 1956.
- [8] Bohm, D. and Nichol, L., *On Dialogue*, Brunner-Routledge, 1996.
- [9] Clayton, R., Rugaber, S., and Wills, L., "Dowsing: A tool framework for domain-oriented browsing of software artifacts", in Proceedings of 13th IEEE International Conference on Automated Software Engineering (ASE '98), October 1998, pp. 204 -207.
- [10] 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.

- [11] Dalton, J. and Smitd, D., *Extending Children's Special Abilities: Strategies for Primary Classrooms*, Melbourne, Victorian Ministry of Education, 1986.
- [12] Duncker, K., *On Problem Solving*, Washington, The American Psychological Association, 1945.
- [13] Ericsson, K. and Simon, H. A., *Protocol Analysis: Verbal Reports as Data*, Cambridge, Mass, MIT Press, 1993.
- [14] Frohmann, B., "Discourse analysis as a research method in library and information science", *Library and Information Science Research* no. 16, 1994, pp. 119-138.
- [15] 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.
- [16] 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.
- [17] 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.
- [18] 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.
- [19] 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.
- [20] Lloyd, P., Lawson, B., and Scott, P., "Can concurrent verbalization reveal design cognition?" *Design Studies*, vol. 16, 1995, pp. 237-159.
- [21] Meijer, J. and Riemersma, F., "Analysis of solving problems", *Instructional Science*, vol. 15, 1986, pp. 3-19.
- [22] Microsoft, "Microsoft Producer for Microsoft Office Powerpoint 2003", <http://www.microsoft.com/windows/windowsmedia/technologies/producer.aspx>,
- [23] Newell, A. and Simon, H. A., *Human Problem Solving*, Englewood Cliffs, New Jersey, Prentice Hall Inc., 1972.
- [24] Olson, G. M., Olson, J. S., Carter, M. R., and Storosten, M., "Small group design meetings: an analysis of collaboration", *Human Computer Interaction* vol. 7, no. 4, 1992, pp. 347-374.
- [25] 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.
- [26] Parnas, D. L., "The limits of empirical studies of software engineering", in *Proceedings of 2003 International Symposium on Empirical Software Engineering*, 2003, pp. 2-5.
- [27] 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.
- [28] Piaget, J., *The Construction of Reality in the Child*, New York, Basic Books, 1954.
- [29] 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.
- [30] Robillard, P. N., Detienne, F., d'Astous, P., and Visser, W., "Measuring cognitive activities in software engineering", in *Proceedings of 20th International Conference on Software Engineering*, 1998, pp. 292-300.
- [31] 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.
- [32] Rosenthal, R., *Experimenter Effects in Behavioral Research*, New York, NY, Appleton Century Crofts, 1966.
- [33] 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.
- [34] Shepherd, A., "Hierarchical task analysis and training decisions", *Programmed Learning and Educational Technology* no. 22, 1985, pp. 162-176.
- [35] 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.
- [36] 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.
- [37] Watson, J. B., "Is thinking merely the action of language mechanism?" *British Journal of Psychology*, vol. 11, 1920, pp. 87-104.
- [38] Xu, S. and Rajlich, V., "Cognitive process during program debugging", in *Proceedings of Third IEEE International Conference on Cognitive Informatics*, Victoria, BC, August 16-17 2004, pp. 176-182.
- [39] 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, CA, 2005, pp. 340-349.