An Open-Source Analysis Schema for Identifying Software Comprehension Processes

Michael P. O'Brien
Department of Information Technology
Limerick Institute of Technology
michael.obrien@lit.ie

Teresa M. Shaft
Michael F. Price College of Business
University of Oklahoma
tshaft@ou.edu

Jim Buckley
Department of Computer Science & Information Systems
University of Limerick
jim.buckley@ul.ie

Keywords: POP-IV.A. *top-down/bottom-up* POP-V.B. *protocol analysis*

Abstract

This paper presents an analysis schema for talk-aloud protocols, which distinguishes between bottom-up and two variants of top-down comprehension (as originally described by Brooks and Soloway et al). The first is 'expectation-based' comprehension, where the programmer has pre-generated hypotheses of the code's meaning before studying the code segments. The second variant is 'inference-based' comprehension, where the programmer derives hypotheses from clichéd implementations in the code.

The schema is placed in the context of other talk-aloud analysis schemas. It is described in detail, allowing other researchers replicate the protocol and assess the generality of its findings. A brief description of an experiment, which uses this schema, is also included.

Introduction

Talk-aloud is used in many studies of software comprehension processes (von Mayrhauser, 1999), (Pennington, 1987), (O'Brien & Buckley, 2001). However, this technique has a number of problems, one of which is the difficulty in comparing the results obtained across experiments (von Mayrhauser, 1999). Von Mayrhauser (1999) states that this is due to the lack of a common, extensible classification schema for talk-aloud protocols. However, given the difficulty in translating talk-aloud in a standard manner (Buckley & Cahill, 1997), such a classification framework also requires formal standardised and open analysis procedures for the talk-aloud protocols captured.

This is highlighted in the evaluation framework proposed in (Good, 1999) for analysis schemas. A number of questions are posed which refer to the importance of a formal, standardised analysis. For example:

- 'Does the schema come with a coding manual, which describes how to apply it?'
- 'Are examples provided for each category in the schema?'
- 'If more than one coder is used, was the reliability reported?'.

In addition, it seems likely that the need for standardised, formal analysis procedures will increase in importance as the study of cognitive processes proceeds. As more subtle comprehension processes are being probed, the analysis schema used to distinguish these categories will become more

involved. Under such circumstances, replication of results will prove impossible without a formal open analysis procedure.

This paper reviews the classification schemas that have been used in software comprehension experiments. This review illustrates that while a number of analysis have been defined, few have been explicitly published, and that explicit analysis protocols for more subtle classification schemas are rare.

The paper also reports on an open, standardised analysis for talk-aloud protocols. This schema extends previous classification schemas of bottom-up and top-down comprehension processes (Shaft, Coders Manual) into three classification categories. It includes the analysis procedure used to translate verbal data into these categories. Being open, it allows for subsequent refinement by other researchers and for replication of experiments.

The schema classifies utterances into three cognitive processes. The first, based on the model proposed by Brooks (1983), is expectation-based comprehension, which, in essence, can be considered a cycle of programmer hypothesis, scanning code for that hypothesis, and validating its existence. The second, as described by Soloway (1984) proposed a slightly different model of inference-based comprehension, where the programmer scans the code, and from incomplete knowledge of that code, infers an abstracted hypothesis. This is subsequently validated against the code. Finally the schema identifies bottom-up processing (Schneiderman & Mayer, 1979), (Pennington, 1987), which can be characterised by an initial study of segments of code, line by line, leading to aggregation, and finally to a conclusion about the purpose of the segment.

A framework for analysis schemas as presented by (von Mayrhauser & Lang, 1999) is described and the analysis schema is detailed in this context. Finally, a short description of an experiment carried out with this analysis schema is presented. This illustrates how the three comprehension processes are related to programmers' familiarity with the domain.

Talk-aloud in Software Comprehension

The Talk-Aloud Protocol

The talk-aloud technique involves a subject completing a task or solving a problem while verbalising his or her thought processes (Nunan, 1992). This entails subjects saying everything that comes into their minds as it comes into their minds, during an experiment session. Talk-aloud has its roots in the area of cognitive psychological research, especially in the areas of chess playing, and mathematics, but has widely been used to evaluate the comprehension processes employed by computer programmers, when comprehending code. Subjects are encouraged to speak out while understanding the code, allowing their utterances to be captured and subsequently analysed.

Effective use of the talk aloud protocol

For many years, the talk-aloud protocol was viewed with suspicion by many empirical researchers (Nisbett & Wilson, 1977), (Lashley, 1923). However, in 1980, Ericsson and Simmons (1980) reported on how the technique is most effectively used. They argue that if the verbal data is produced concurrently and if the subjects' report on what enters their mind, rather than on their processes, then the data produced is reliable. Under such conditions, talk-aloud technique provides the "richest source of a programmer's mental state" (Russo, Johnson, & Stephens, 1989).

However, reliable data, and reliable finding are not the same thing, and care must be taken to translate this data into a form that provides reliable information on programmers' cognitive processes.

Talk-aloud Analysis Schemas in Software Comprehension Studies

This section reviews some of the studies carried out in software comprehension with respect to their talk-aloud analysis schemas. It describes the classification schemas employed and the analysis protocols used to analyse the data.

Letovsky (1986) used concurrent verbal data from programmers who were involved in maintaining a small database system. In his study he identified distinct bottom-up and top-down comprehension episodes (categories) in the talk-aloud data. He also identified a number of other cognitive processes in the protocols such as plausible slot-filling and abduction (Eysenck & Keane, 2000).

His analysis procedure was to identify questions in the talk-aloud protocol like 'How did the programmer change the order?', 'What does this code segment do' and 'Why did the programmer do this?' The 'How' questions were considered indicative of top-down processing where the subject is trying to figure out how a goal is achieved in the code. The 'What' questions and the 'Why' questions were considered indicative of bottom-up processing as the subject was trying to figure out what the code does, and why particular code is present (i.e. what goal does it achieve). Letovsky gave extensive examples to support this analysis procedure. However, for the other cognitive processes suggested by Letovsky, he presents only one anecdotal episode.

Pennington's classification schema (Pennington, 1987) was originally for written summary analysis and was later extended by Good (1999). These retrospective summaries can be considered analogous to retrospective talk-aloud data. It involved two separate analyses on program summaries, classifying each statement by both, 'Information Type' and 'Level of Detail' (Pennington, 1987). Subjects in Pennington's experiment were presented with a program of moderate length, and were asked to write two summaries of the program itself. The first summary was requested after 45 minutes, and the other, after carrying out a small modification to the program. It was these summaries that were classified by type and detail. The information types investigated here were:

- "Procedural Statements include statements of process, ordering, and conditional program actions". (Pennington, 1987)
- Data Flow Statements are statements about data structures, and data manipulation.
- Functional statements are statements about the function of the program in terms of the real world domain.
- Vague statements are statements, which do not fit into any of the above 3 categories. Good (1999) calls this a bucket category, acting as a hold all for statements, which do not fit, into the other categories.

Pennington specified four levels of detail for coding in her second classification schema: -

1 Detailed statements refer to the program's operations and variables

2 Program statements refer to a program's procedural blocks, for example, a search routine.

3 Domain statements refer to real world objects, such as in the previous example in this section, cables or buildings.

4 Vague statements don't have specific referents

Pennington does not report a formal analysis protocol. However, Good's thesis suggests that she may have (Good, 1999). Indeed the categories seem intuitively easy to distinguish, depending, as they do, on the objects to which the subjects refer, and on the orthogonal information types.

Shaft's research demonstrated the role of application domain knowledge to the comprehension process of computer programs (Shaft, 1995). She distinguished between top-down and bottom-up and, in carrying out her experiment, developed a 'Coders' Manual' to assist the process of coding the verbal data. She ensured translation objectivity by employing two independent coders for her

analysis. She stated clearly that the role of the coder is to code the verbal protocols according to the schema contained in the coding manual. In doing so, Shaft identified the distinct role of the coder.

Coders were presented with a hardcopy of the program used in the experiment, all relevant documentation, the manual and a transcript of the verbal data protocols themselves. After training, they are asked to assign codes to the statements in the protocol. Statements were assigned a code to represent top-down comprehension or bottom-up comprehension.

In the manual, Shaft presented the characteristics of bottom-up episodes and top-down episodes. She re-enforced this with a number of examples and finally gave a decision tree to facilitate coders' identification of the relevant episodes.

She also identified three reasons why certain phrases are left uncoded. Firstly, the phrase may not belong in one of the other categories. Secondly, phrases may combine with statements above or below it, and finally, phrases made by the subject to the experimenter should not be coded.

Von Mayrhauser (1994) classified verbal data produced from her experiments into the following three models, which collectively, make up her proposed meta-model (von Mayrhauser, 1995): -

- Top-down model
- Situation model
- Program model

Von Mayrhauser described her analysis procedure as consisting of three steps; (1) enumeration of action types, (2) segmentation of the protocols and identification of information and knowledge types, and (3) deduction of dynamic code understanding processes, by classifying and analysing episodes. She did not expand on how to carry out this classification.

A Classification Schema Framework

Von Mayrhauser & Lang (1999) state that although protocol analysis is a very valuable tool in the area of empirical studies, it has several drawbacks. One is the difficulty of comparing results across different studies, and this led them on to develop their Flexible Expandable Coding Schema (AFECS). This coding schema aims to provide an extensible standardised classification schema for talk-aloud protocols. They state that many observational studies use unique coding schemas, making comparisons between studies difficult to achieve. The AFECS coding schema can be expanded to the task being observed, and maintains a high degree of standardisation, enabling comparisons to be made across studies.

AFECS is designed so that each classification (code) explicitly identifies each programmer action, along with the objects of that action. Codes are split into a number of segments, represented by levels in Fig 1. Each segment represents an aspect of a cognitive process. These 'segments' are levelled by granularity and specificity. AFECS classes each part of a programmer's protocol into a maximum of six segments and into a minimum of 3. The resultant code assigned contains at least a verb, noun, and adjective (see Fig. 1).

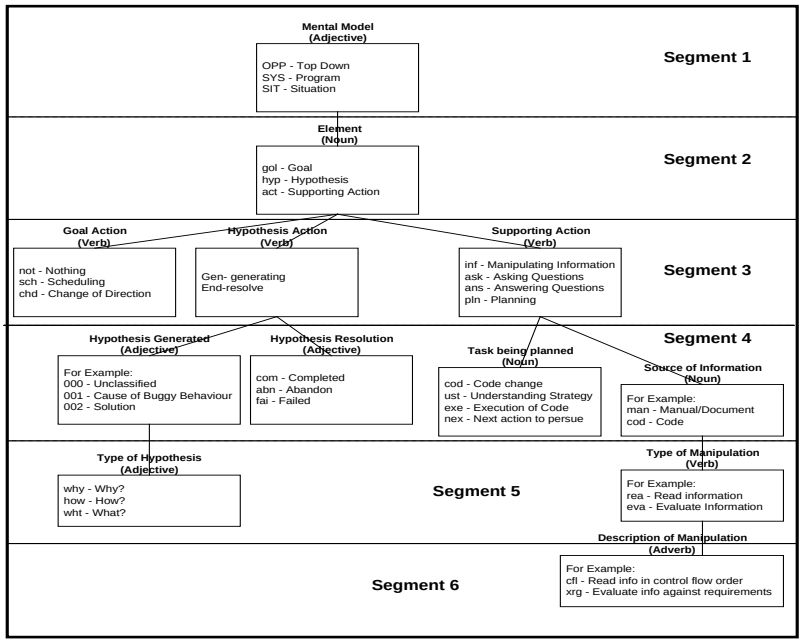

Figure 1 AFECS Coding Schema

The first two segments (mental model, and element) of the coding schema are mandatory. The remaining optional segments provide more detail for a specific value of a higher-level code segment. For example, if a hypothesis were resolved, one would code ‘end’ in box ‘hypothesis action’.

Von Mayrhauser et al (1999) state that the main advantages of splitting codes into segments include: -

- Flexibility
- Expandability
- Ease of manual coding
- Ease of automation in analysing code sequences

Other talk-aloud classification schemas can fall into and expand this framework.

Refining a schema for Top-down Comprehension

Introduction

The purpose of this research is to investigate the cognitive processes Information Systems professionals’ use when they attempt to understand computer programs. To provide a trace of their thought process, programmers are asked to “think aloud” while studying computer programs. A tape recorder is used to capture their verbalisations. Each programmer’s verbalisations are transcribed into short, independent phrases, providing a measurement of the information used by the programmer. Fig. 2 contains one page taken from a protocol of a subject studying a computer program. After the protocols have been transcribed, coders classify the phrases or statements according to a pre-determined coding scheme contained in a coders manual (O’Brien & Buckley, 2000). Formal statistical analysis is conducted on the coded data.

The role of the coder is to code the protocols according to the coding scheme and coding procedure discussed in detail below. Since the output of the coding processes becomes the input to formal analysis, reliability is an essential issue. Therefore, two different coders working independently

coded each protocol. The results were compared (by the coders) to resolve disagreements and when they were finished the results were returned for analysis.

As in Shaft's work, coders were given a hard copy listing of the computer programs used during the experiment. The listing contained a cross-reference list of all file names, data names, and paragraph names in the program. Coders were given the coders manual, which described in detail the analysis process to be carried out. This is discussed in section 3.2. They were also given the transcripts of talk-aloud protocol data as shown in Fig. 2.

When the coders were given the coding manual and were satisfied that they understood the coding rules, they were given some complete protocols to code as a practice assignment before the experiment proper.

When coders finished coding a protocol for a subject they were asked to set it aside for a short time. After a break, they checked the codes they had assigned. When they were satisfied that the codes had been properly assigned, they then returned the coding sheets and protocols to the researcher.

```
ACCOUNTING – COMPREHENSION
SUBJECT ID: SUBJECT1.ACC
PHRASES: 81 TO 100

81.     CHECK-TWENTY-ONE-OUT
82.     This looks like ID numbers
83.     Or looks like we're printing a check
84.     HEADLINE
85.     REPORT-NAME,
86.     WORK-STUDY ROSTER
87.     TITLELINE
88.     NAME,          SOCIAL-SECURITY-NUMBER,
        BUDGET, GROSS-PAY, NET-PAY, LOCATION-
        LINE, DATE, LOCATION, PAGE
89.     Next page
90.     DEPARTMENT-LINE
91.     TOTALS FOR DEPARTMENT, NUMBER OF
        STUDENTS
92.     DEPARTMENT-CT, looks like a count
93.     DEPT-GROSS-AC,    DEPT-NO,    DEPT-TYPE,
        something to do with ordering perhaps
94.     DEPT-NET
95.     And there are flags beside the COUNT, the GROSS,
        and the NET
96.     UNIVERSITY-LINE looks like the same thing,
97.     GRAND TOTAL
98.     So up above we have looks like a DEPARTMENT-
        LINE and then the UNIVERSITY-LINE is a
        GRAND-TOTAL-LINE
99.     MAILING is the LABLES bit
100.    FIRST-NAME,
```

Figure 2: Excerpt from a protocol of a subject studying a COBOL program

3.2 Coding Categories

3.2.1 Definition of Code Categories

The purpose of this analysis schema is to determine the type of comprehension process programmers use to understand computer programs. Programmers may use a top-down or bottom-up process. Using a top-down process, programmers' may start with a general hypothesis about the nature of the program and seek to refine their understanding of the program by proposing and testing increasingly specific hypotheses (Brooks, 1983). One top-down process is evidenced by hypotheses based on knowledge previously obtained by the programmer and originating from their own knowledge base (Brooks, 1983). A distinct top-down comprehension process is observed when the programmer makes a statement about the nature of the software based on limited information from the actual program (Soloway, 1984). It is in effect a "hunch", or reasoned guess. The third category is bottom-up comprehension, which occurs when the programmer examines the program directly line-by-line, building up to a general understanding of the program.

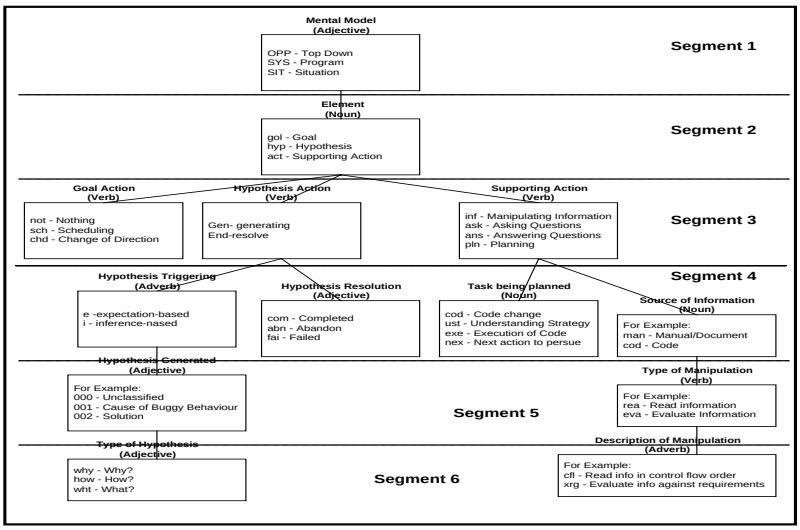

Figure 3 AFECS Coding Schema (Expanded)

Hence this analysis schema aims to identify these three different comprehension episodes: pre-generated hypotheses episodes, inference- based hypotheses episodes and bottom-up episodes.

This schema fits into the AFECS framework under the 'Hypothesis Action' box in segment 3 as shown in Fig. 3. If the hypothesis action is to generate a hypothesis, then the newly inserted box describes the trigger for this generation process. The boxes 'Hypothesis Generated' and 'Type of Hypothesis' are relevant for both triggering processes and so are placed below the newly inserted box.

The analysis procedure defined is to examine each phase and determine if it is a generalisation about the nature of the program or some portion of the program. If so, it is necessary to determine if it is a *Pre-generated Hypothesis*, an *Inference-based Hypothesis*, or part of *Bottom-up Comprehension*, using the criteria presented below. If a phrase cannot be classed as a Pre-generated Hypothesis, an Inference-based Hypothesis, or Bottom-up then no code is assigned.

Identifying Expectation-Based Hypotheses

A hypothesis may be categorised as pre-generated if the subjects' use the definite article, or utter phrases like, "I expect".

```
/* Code Segment */
for (i=0; i<n-1; i++) {
  for (j=0; j<n-1-i; j++)
    if (a[j+1] < a[j]) {
      tmp = a[j];
      a[j] = a[j+1];
      a[j+1] = tmp;
  }
}
```

Figure 4 – C code implementation of a Bubble Sort

From Fig. 4, examples of expectation-based hypotheses from verbal data protocols would include, "*I expect* to see a bubble sort…", or "Where is a section that deals with *the* sorting…". The definite article suggests it was something that the programmer was waiting for.

Expectations may also be classified by words which suggest omission, such as "where is", "there should be", and "normally". For example, programmers might say, "there *should be* some sort here", "there is *normally* some sort procedure….".

3.2.1.2 Identification of Inference-based Hypotheses

Inference-based hypotheses are identified by the use of the indefinite article and by a level of uncertainty in the hypothesis. The indefinite article suggests that it was not pre-expected by the programmer and the level of uncertainty suggests that it has not been deduced emphatically by a bottom-up study of the code. Examples of inference-based hypotheses from the code in Fig. 4 include: - "Oh right, there *seems* to be *a* sort here..." or "I *guess* there is *a* sort of some kind here which involves a swap." Phrases like 'that's *probably a*', also suggest that the hypothesis is code-prompted.

3.2.1.3 Identification of Bottom-up Comprehension

Bottom-up comprehension can be interpreted as a line-by-line description of code leading to the eventual deduction of plan. It is characterised by a build-up of knowledge and a fairly high level of certainty in the conclusion

Again, using the code in Fig. 4, an example of a bottom-up phrase may be: ″if a[j+1] < a[j]. Okay, here we see if a[j+1] is less than a[j]. If it is, we move a[j] to a temporary variable called 'temp'. a[j] is then equal to a[j+1] and a[j+1] is equal to temp. *This is some kind of array swap"*

The following cautionary notes were included in the analysis schema for when coders were coding Bottom-up Comprehension:

- *Do not code translations*. Phrases, which are simple translations of program statements, for example, from C to English, are not classified as bottom-up Comprehension and should not be coded (see Fig. 5).

```
209  ...
210  if a[j+1] < a[j]
211  okay, here we see if a[j+1] is less than a[j]
212  ...
```

Figure 5 Verbal Protocol data

- *Do not code phrases where the programmer summarises his/her understanding of the program*. Consider the situation where a programmer reads back through a portion of the program. In these situations a programmer may refer to a paragraph in a summary fashion, making statements such as, in this example, "Print off monthly totals" when referring to the FINISH-MONTH paragraph in a COBOL program, per say. Such phrases should not be coded since the programmer does not actually comprehending at this time, but is restating what they

```
216  ...
217  if ((i%2) != 0)
218  this is seeing if i mod 2 is not equal to 0
219
```

already understood. Note the phrases in Fig. 6, where the programmer is paraphrasing and summarising his/her understanding of the program.

Figure 6 Excerpt from a talk-aloud protocol

Good (1999) emphasises the importance of a formal decision process for the coders. The decision process detailed for the coders in this analysis schema was:

- *Is the statement the critical statement within the episode?* If your answer is "yes", (i.e. the statement is a summary of more than one line of code), then continue to question 2. If your answer is "no", then no code should be assigned, and you should continue to the next phase of the protocol.
- *Does the phrasing of the statement indicate a hypothesis*? If the critical statement contains words such as "that's probably", or "this is maybe", with the indefinite article, then assign a code of "I", indicating an Inference.
- *If the critical statement contains* phrases such as "I expect", or "where is", with the definite article, then assign a code of "G", indicating a Pre-generated expectation.
- *Is the critical statement preceded by a full build up of knowledge and phrased as a conclusion*? If the critical statement meets both of these conditions, assign a code of "B", indicating bottom-up comprehension. Bottom-up comprehension must be preceded by a build-up of knowledge culminating with a conclusion of some sort. If the critical statement does not meet both of these conditions, continue to question 4.
- *Is there no full build up of knowledge prior to the critical statement*? To be classified as bottom-up comprehension, the statement must be preceded by evidence of a full build-up of knowledge. If such a build-up is lacking, but the phrase is clearly a critical statement, assign a code of "I", for Inference.

3.2.2 Uncoded Phrases

Three reasons to leave a phrase uncoded were described to the coders:

1 *It may not belong in one of the coding categories*. This will not be unusual. This research investigates specific questions. Therefore only certain kinds of information are of interest (see Definition of Code Categories). For instance, programmers often read the computer program aloud. Also, phrases that correspond to paraphrasing the program should not be coded.

2 *The phrase may combine with statements above or below it*. If two phrases are parts of the same critical statement, they should not be coded separately. Instead, the appropriate code should be assigned to the line number of the last phrase.

3 Any phrase made by the subject to the experimenter should not be coded.

The recommended procedure is to examine each phrase and determine if it should be coded as pre-generated, inference-based, or bottom-up, or indeed, left uncoded. If the phrase cannot be classified, coders do not assign a code for any category and continue on to the next phase.

The Experimental Study

As described above, the study attempts to illustrate the existence of pre-generated expectations and inferences in programmers' comprehension sessions and to examine the effect of domain familiarity on these processes. It is hypothesised that programmers, who are familiar programming in a particular domain to particular standards, will rely, more, on pre-generated expectations & inferences than those who are unfamiliar with the domain (see Fig. 7).

On the other hand, it is proposed that programmers' unfamiliar programming in a particular domain will have to rely, more, on bottom-up processes. However, if they possess a detailed knowledge of the programming language at hand and programming standards, they may rely somewhat on inferences, but certainly, will rely less on expectations. This is because they, being unfamiliar with the domain, will not have many expectations.

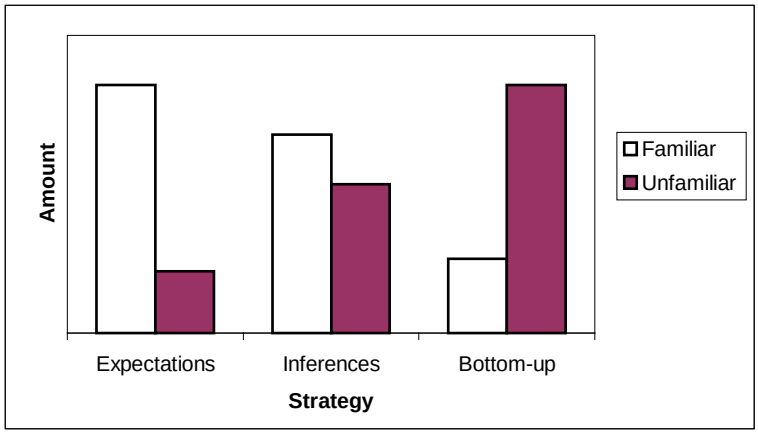

Figure 7 – A hypothesis of the relative frequency of each strategy to be expected in Familiar & Unfamiliar Domains

Eight professional/industrial programmers from two independent software companies (4 from a healthcare company and 4 from an insurance company) participated in this experiment. They were presented with two programs, one from the healthcare domain and the other from the insurance domain. Their verbal protocols were analysed by two independent coders using the coding schema described in section 3.2.

4.1 Reconciled Results

Results obtained from this experiment suggested a high level of agreement between the two coders. Both coders had a very similar number of pre-generated hypothesis episodes, inference episodes and bottom-up episodes. However, to ensure that the same phrases were being coded consistently, the individual coded transcripts were viewed and the categorisation of each phrase was assessed for agreement between each coder. The results are presented in Table 1.

	Coder 1	**Coder 2**
Consistent	108	108
Inconsistent	5	5

Table 1 – Coders' consistency results per phrase (all categories)

This table identifies the amount of phrases, which were coded consistently and inconsistently. It indicates that they were in agreement over 95% of the time. However this figure consists of bottom-up phrases as well. Just considering the more subtle distinction of pre-generated episodes and inference-based episodes, the figures are presented in Table 2.

	Coder 1	**Coder 2**
Consistent	85	85
Inconsistent	4	4

Table 2 – Coders' consistency results per phrase (in terms of expectation-based & inference-based)

Here a slight drop in consistency is shown, but still, the coders agreed for approximately 93% of the phrases. This indicates that the two cognitive processes are distinct and can be reliably distinguished.

Domain	Expectations	Inferences	Bottom-up
Familiar	20	31	20
Unfamiliar	2	36	4

Table 3 – Coders' reconciled results

The results given in Table 3 support, in part, the experimental hypothesis. That is, there seems to be a relationship between programmer familiarity and the pre-generated expectations a programmer will employ. However, the expected relationship between familiarity and bottom-up comprehension was not present. This is a surprising finding, which is discussed more fully in (O'Brien & Buckley, 2001).

Conclusion

This paper illustrated the importance of analysis procedures for instruments like talk-aloud. It presented one such procedure and its associated categories, along with briefly describing a small experiment to show its usage. The paper also examined several classification schemas used by various researchers in the area of empirical studies of programmers. In doing so, the authors have stressed the importance of open standardised analysis schemas for identifying software comprehension processes.

Acknowledgements

This research was funded by Enterprise Ireland.

References

Brooks, R., (1983) Towards a Theory of the Comprehension of Computer Programs. *International Journal of Man-Machine Studies*, Vol. 18

Buckley, J., Cahill, A., (1997) Measuring Comprehension Behaviour through System Monitoring. *Proceedings of the Workshop on Empirical Studies of Software Maintenance*

Ericsson, K., Simmons, H., (1980) Verbal Reports as Data. *Psychological Review*, Vol. 89, No. 3

Eysenck, M., Keane, M., (2000) Cognitive Psychology: A Student's Handbook, *4th Edition*, Psychology Press

Good, J., (1999) Programming Paradigms, Information Types and Graphical Representations: Empirical Investigations of Novice Program Comprehension, *Ph.D. Thesis*, University of Edinburgh.

Lashley, K., (1923) The Behaviouristic Interpretation of Consciousness II. *Psychological Review*, Vol. 30.

Letovsky, S., (1986) Cognitive Processes in Program Comprehension, *Empirical Studies of Programmers: 1st Workshop*.

Nisbett, R., Wilson, T., (1977) Telling more than we can know: Verbal Reports on Mental Processes. *Psychological Review*, Vol. 84.

Nunan, D., (1992) Research methods in language learning. Cambridge University Press.

O'Brien, M., Buckley, J., (2001) Inference-based & Expectation-based Processing in Program Comprehension. *To appear at the 9th International Workshop on Program Comprehension*, May 2001.

O'Brien, M., Buckley, J., (2000) The GIB Talk-aloud Classification Schema. *Technical Report 2000-1-IT*, Limerick Institute of Technology, (Available on request from authors).

Pennington, N., (1987) Comprehension Strategies in Programming. *Empirical Studies of Programmers: 2nd Workshop.*

Russo, J., Johnson, E., Stephens, D., (1989) The Validity of Verbal Protocols, *Memory & Cognition*, Vol. 17.

Schneiderman, B., Mayer, R., (1979) Syntactic / Semantic Interactions in Programmer Behaviour. *International Journal of Computer and Information Sciences*, Vol. 8, No. 3.

Shaft, T. M., "*Coders Manual*", Source: Personal Correspondence.

Shaft, T. M, (1995) The Relevance of application domain knowledge: the case of computer program comprehension. *Information Systems Research*, Vol. 6, No. 3.

Soloway, E., (1984) Empirical Studies of Programming Knowledge. *IEEE Transactions on Software Engineering*, IEEE Computer Society, Vol. SE-10, No. 5

Von Mayrhauser, A., Lang, S., (1999) A Coding Scheme to Support Analysis of Software Comprehension. *IEEE Transactions on Software Engineering*, Vol. 25, No. 4, July/August.

Von Mayrhauser, A., Vans, A., (1994) Dynamic Code Cognition Behaviours for Large Scale Code. *Proceedings of 3rd Workshop on Program Comprehension*

Von Mayrhauser, A., Vans, A., (1995) Program Understanding: Models & Experiments, *Advances in Computers*, Vol. 40.