JOURNAL OF SOFTWARE MAINTENANCE AND EVOLUTION: RESEARCH AND PRACTICE
*J. Softw. Maint. Evol.: Res. Pract.* 2004; **16**:427–447
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/smr.307

**Research**

# Expectation-based, inference-based, and bottom-up software comprehension

Michael P. O'Brien[1,*,†], Jim Buckley[1] and Teresa M. Shaft[2]

[1]*Department of Computer Science and Information Systems, University of Limerick, Ireland*
[2]*Michael F. Price College of Business, University of Oklahoma, U.S.A.*

**SUMMARY**

**The software comprehension process has been conceptualized as being either 'top-down' or 'bottom-up' in nature. We formally distinguish between two comprehension processes that have previously been grouped together as 'top-down'. The first is 'expectation-based' comprehension, where the programmer has pre-generated expectations of the code's meaning. The second is 'inference-based' comprehension, where the programmer derives meaning from clichéd implementations in the code.**

**We identify the distinguishing features of the two variants, and use these characteristics as the basis for an empirical study. This study establishes the existence of the above-mentioned processes, in conjunction with 'bottom-up' comprehension. It also illustrates the relationship between these processes and programmers' application domain familiarity. Copyright © 2004 John Wiley & Sons, Ltd.**

KEY WORDS: software comprehension; cognitive strategies; top-down comprehension; program understanding

## 1. INTRODUCTION

Program comprehension is a major part of software development, especially during the maintenance phase. Studies have shown that up to 70% of lifecycle costs are consumed within the maintenance phase [1] and that up to 50% of maintenance costs relate to comprehension alone [2]. Hence, according to these studies, up to 35% of the total lifecycle costs can be directly associated with understanding the original computer program.

*Correspondence to: Michael P. O'Brien, Department of Computer Science and Information Systems, University of Limerick, Ireland.
†E-mail: michaelp.obrien@ul.ie

Contract/grant sponsor: EI Basic Research Project; contract/grant number: SC\1999\035\Y
Contract/grant sponsor: Science Foundation Ireland Investigator Programme, B4-STEP (Building a Bi-directional Bridge Between Software ThEory and Practice)

*Received 16 April 2003*
*Revised 1 October 2004*
*Accepted 4 October 2004*

Research into software comprehension models suggests that programmers attempt to understand code using bottom-up comprehension, top-down comprehension, and various combinations of these two processes. Bottom-up comprehension models propose that, as source code is read, abstract concepts are formed by chunking together low-level information [3,4]. Linger *et al.* [5] referred to bottom-up comprehension as stepwise abstraction, where a programmer forms an abstract description of a program fragment's goal from a methodical study of the fragment itself. Schneiderman and Mayer [3] suggested that this bottom-up approach is based on the constraints of humans' short-term or processing memory [6]. Pennington [4] expanded on Linger *et al.*'s theory by suggesting that programmers understand unfamiliar code by firstly attempting to form a 'program model', in which the control-blocks (while loops, for loops, if constructs, etc.) of the program are abstracted into mental 'chunks'. Following the partial construction of this 'program model', the programmer begins to create a more domain-oriented model of the system, which Pennington refers to as the 'situation model'. This construction maps the (possibly delocalized) events, states and structures of the source code to the events, states and structures of the real world [7]. This bottom-up process suggests a staged model of comprehension. However, the order in which these activities occur may depend on factors such as the program structure or the language paradigm used [8]. To the best of our knowledge, the influence of these factors has not been well established [9].

Letovsky [10] identified episodes of bottom-up processing in practice, when he noticed programmers asking 'what' and 'why' type questions. The 'what' questions reflected the programmers' need to identify 'what' the code was doing and the 'why' questions reflected their need to identify 'why' certain code was used. Both suggest a bottom-up approach to comprehension and this has been further validated in industrial contexts by [11,12].

The bottom-up models of software comprehension address situations where the programmer is unfamiliar with the application domain. Several 'top-down' models of software comprehension have been developed to address the alternative situation where the programmer has some previous exposure to the application domain. By application domain, we refer to the context of the problem that is addressed by a piece of software. As such, we distinguish the application domain from the programming domain, which concerns itself with technical details of implementing the solution [13].

Top-down models of comprehension propose that the programmer utilizes application domain knowledge to build a set of expectations that are mapped on to the source code [14–16]. We describe these models in more detail in Section 1.1.

It is unlikely, however, that programmers exclusively rely on either one of these strategies. Instead, empirical research suggests that they subconsciously adopt one of these to be their predominant strategy, based on their knowledge of the application domain under study [12,15,17], and switch between comprehension processes as cues become available to them [18]. In fact, Letovsky refers to programmers as 'opportunistic processors' to reflect the ease with which they change their comprehension strategies in response to external cues and stimuli [10].

### 1.1. Background

We specifically distinguish between the two models of comprehension defined by Brooks [14] and Soloway and Ehrlich [19]. In the past, these two models have been grouped together as one top-down process, whereas, when studied in depth, they appear to differ [20].

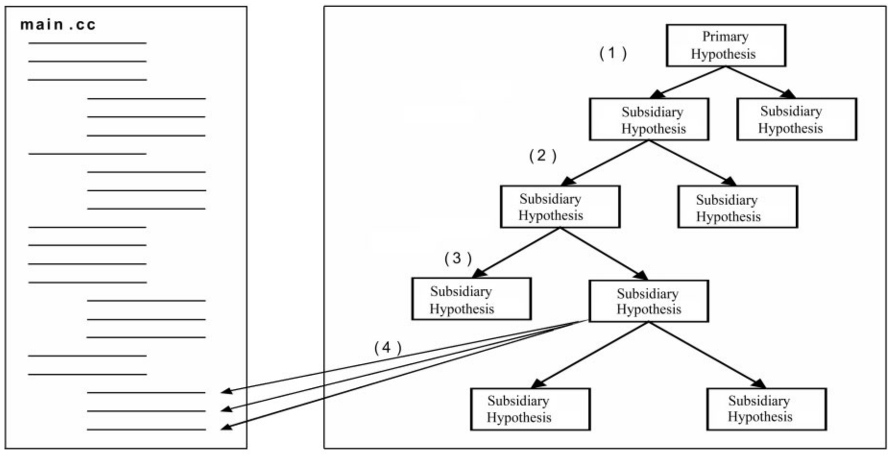

Figure 1. Brooks' model of program comprehension.

Brooks presented a theory of top-down comprehension in which programmers' pre-generate hypotheses (see step 1 in Figure 1). These hypotheses may come from other programs, expertise, or documentation. According to Brooks, hypotheses are subsequently subdivided into finer hypotheses (steps 2 and 3 in Figure 1) until they can be bound to particular code segments (see step 4 in Figure 1), and thus be validated.

The starting point for Brooks' theory is an analysis of the structure of the knowledge required when a program is understood. This theory views knowledge as being organized into distinct domains that bridge between the original problem and the final program [13]. The program comprehension process is one of reconstructing knowledge links between these domains for the programmer [14]. This reconstruction process is theorized to be top-down and hypothesis driven, when the programmer is familiar with the application domain. An initially vague and general hypothesis is refined and elaborated upon based on the programmer's knowledge, information extracted from the program text and other relevant documentation. Brooks proposes that 'beacons' in the code are used to confirm or repudiate hypotheses or expectations. Essentially, beacons are stereotypical segments of code that indicate the program's functionality [21]. Brooks did not carry out any formal empirical evaluation of this theory, although several researchers have since performed empirical studies that support it [12,15,18].

Soloway's work [19], although often interpreted as a model that is exclusively top-down [18,22,23], has both top-down and bottom-up elements in it. Soloway proposes that programmers initially scan the code using 'shallow reasoning', for critical line(s) or beacons. These beacons suggest the presence of plans in their source code. Plans are 'program fragments that represent stereotypical action sequences in programming to achieve a specific goal' [24]. After finding the beacon, programmers seek to validate the presence of the goal, by studying the code for additional elements of the plan. From Soloway's empirical study there was no evidence that Brooks-type pre-expectations were employed as participants had no prior knowledge of the program or its application domain before they started their sessions (and thus could have no pre-expectations).

Essentially then, the differing characteristic between the Brooks' and Soloways' models, *is what triggers plans in programmers' minds when understanding systems*. Brooks suggests that the trigger is pre-generated hypotheses and that beacons are used to confirm the existence of these plans. Soloway's studies suggest that the beacons in the code act as the trigger, which results in a hypothesis that a plan is present within the code.

Letovskys' view of programmers as 'opportunistic processors', suggests that the truth possibly lies between the two. Programmers may scan the code, find a beacon, and infer a plan, which then sets up expectations for some other related plans in a 'plan hierarchy/semantic net'. While searching for these pre-generated plans, the programmer may come across another beacon and infer another unexpected plan.

### 1.2. Paper structure

In this study, we differentiate between and establish the co-existence of pre-expectation [14] and inference [19] based comprehension. Their co-existence is established by conducting an experiment in which industrial software engineers try to understand software from a familiar and unfamiliar application domain. Participants' talk-aloud protocols are captured and subsequently categorized into expectation-based, inference-based, and bottom-up comprehension by independent coders, to a high degree of reliability. This analysis is based on an *a priori* coding schema [25].

Further, we establish a relationship between application domain familiarity and predominance of comprehension strategy. When a programmer is familiar with the application domain, it is hypothesized that this knowledge allows them to generate more pre-expectations and thus emphasize a Brooks-type comprehension process. Similarly, knowledge of the application domain also allows a programmer to use an inference-based process as they can recognize beacon elements of domain goals. However, when working in an unfamiliar application, a programmer will be forced to rely more on bottom-up processing as they lack the knowledge needed to generate an expectation or identify a beacon.

## 2. THEORETICAL DEVELOPMENT

The primary purpose of this study is to determine if programmers utilize different forms of 'top-down' comprehension processes. Specifically, earlier empirical studies have not distinguished between expectation-based and inference-based comprehension processes. Expectation-based comprehension relies upon programmers' existing knowledge base to generate expectations about the goals of the code in a computer program. Inference-based comprehension occurs when programmers identify a beacon in the code and through further examination confirm their initial interpretation of the beacon. We state the following hypothesis.

**Hypothesis 1.** *Programmers use both expectation-based processing and inference-based processes during software comprehension.*

We also examine the relationship between programmers' existing knowledge and their use of the different comprehension processes (expectation, inference, or bottom-up). Programmers with application domain familiarity possess more knowledge of the domain goals. We argue that, from this knowledge, they are able to generate expectations as to the system's goals. This view is consistent with Brooks' [14] argument that programmers require application domain knowledge to generate a primary-hypothesis about the nature of a computer programmer. Programmers unfamiliar with the application domain will lack this knowledge and, thus, are unable to generate expectations of the system's functionality. Therefore, we state the following hypothesis.

**Hypothesis 2(a).** *When a programmer is familiar with the application domain, they use more expectation-based processing than when unfamiliar with the application domain.*

Inference-based comprehension occurs when programmers recognize a beacon in the code [19]. The ability to identify a beacon suggests that the programmer has identified a plan and thus a system goal. These goals could be algorithmic or application domain goals. When programmers are knowledgeable and experienced with respect to the programming language, as is the case in this study, it is anticipated that they are able to identify beacons for algorithmic goals. However, when working in a familiar application domain, programmers will also be able to identify beacons for application domain goals. Therefore, it is hypothesized that when programmers are familiar with the application domain they will perform more inference-based processing that when they are unfamiliar with the application domain.

**Hypothesis 2(b).** *When a programmer is familiar with the application domain, they use more inference-based processing than when unfamiliar with the application domain.*

Both expectation-based and inference-based comprehension processes are less accessible to programmers who are studying systems from unfamiliar application domains. When programmers are unfamiliar with the application domain, they lack the knowledge necessary to generate pre-expectations. Further, they may rely on inference-based processing only when the beacon is related to their programming knowledge, limiting their use of inference-based processing. Thus, without application domain knowledge, programmers have little alternative than to use bottom-up comprehension processes. Hence, we state the following hypothesis.

**Hypothesis 2(c).** *When a programmer is unfamiliar with the application domain, they use more bottom-up comprehension than when familiar with the application domain.*

## 3. METHODOLOGY

To test the hypotheses, we conducted an investigation of the comprehension processes of professional computer programmers. We required two application domains to test our hypotheses and selected healthcare and insurance. In the next sections we describe participants' backgrounds, the experiment's design and the experiment's procedures.

### 3.1. Participants

Eight professional/industrial programmers from two independent software companies (four from a healthcare company and four from an insurance company) participated in this experiment.

Table I. Participants' backgrounds.

| Characteristics | Healthcare company | Insurance company |
|---|---|---|
| Average age | 25 years | 28 years |
| Grade in programming | 62% | 67% |
| Self assessment of the COBOL language | 3.3/5.0 | 3.8/5.0 |
| Years in insurance | 0.6 years | 5 years |
| Familiarity with insurance domain | 1.2/5 | 3.5/5 |
| Years in healthcare | 2.5 years | 0 years |
| Familiarity with healthcare domain | 3.3/5 | 0/5 |
| Years with company | 2.6 years | 5 years |

Participants were employed in developing and maintaining COBOL systems, and were each given £20 to cover any expenses incurred as a result of their participation. Participants were also given confidential feedback on the comprehension strategies they employed during the comprehension sessions.

A background questionnaire was administered prior to the study to assess participants' programming grades, experience programming in COBOL, their application domain knowledge, and their self-professed knowledge of the COBOL language (see Table I).

### 3.2. Experiment design and procedure

The experiment required participants to comprehend two COBOL programs, each consisting of just over 1200 lines of code, which were presented in hardcopy format. Of the two programs used in the study, one was from the insurance company and the other was from the healthcare company. All programmers studied both programs: hence all participants studied programs from a familiar and an unfamiliar application domain. This was important because, to fully test Hypotheses 2(a), 2(b) and 2(c), we need to assess programmers' comprehension processes in both a familiar and unfamiliar application domain context.

The comprehension sessions took place in three segments, a practice session (code from a generic application domain), and two experimental sessions (code from each application domain). At the beginning of each segment, all participants were given a hardcopy of the source code and asked to study the program. They were to obtain as complete an understanding of the program as the time allowed and were informed that their knowledge would be assessed at the end of their session.

All participants were requested to talk-aloud for the duration of the experiment so that their mental states could be captured. That is, their talk-aloud data was gathered concurrently. The participants were asked to simply report on what came into their head, as it came into their head. If, at any stage, they begin to work silently, they were simply prompted: 'What are you thinking now?' by the experimenter. This is consistent with the recommendations for interacting with subjects during think-aloud data collection as suggested by Ericsson and Simmons [26] and Russo *et al.* [27]. Under such conditions, these authors state that the talk-aloud technique provides the 'richest source' of a programmer's 'mental state'.

Table II. Difficulty of the source code.

| Complexity measure | Healthcare program | Insurance program |
|---|---|---|
| SLOC | 1153 | 1179 |
| Assessor 1 | 3 | 4 |
| Assessor 2 | 3 | 3 |
| Assessor 3 | 3 | 4 |
| Assessor 4 | 4 | 3 |
| Median assessor rating | 3.25 | 3.5 |

Programmers were presented with code from a familiar domain in one session, and unfamiliar domain in another. The order of presentation of the application domains was counter-balanced such that half studied the program from the familiar application domain first, followed by the unfamiliar domain, while the other half of the programmers studied the program from the unfamiliar domain first. At the end of each session, programmers were asked to provide a free format summary of each of the systems they had studied. Between sessions, participants were given a set of puzzle-like exercises to distract their attention from the task previously undertaken (these exercises were taken from the http://www.bored.com Web site). The exercises gave them a chance to clear their minds, so they could start afresh in the next session without the possibility of carryover details from the session just undertaken.

### 3.3. Program source code

The insurance program used in the experiment created a new company insurance policy file using data from an input policy and employee files. The new policy file had to contain all relevant policy and employee details, i.e., it contained only those policies with current or impending effective periods. The healthcare program read an employee file and healthcare policy file, using a joint Social Security Number as a common identifier. From these, it generated a file with active healthcare policies.

It was vital that the programs from the two application domains be comparable on overall complexity. Since many different program complexity measures are highly correlated with source lines of code (SLOC), we used SLOC as one of our complexity standards [28]. We also acquired four professional COBOL programmers to independently assess the complexity of the two programs. They rated the programs on a scale 1–5, where 1 was 'simple' and 5 was 'very difficult'. Results of these ratings are shown in Table II. Both measures show that the programs were of similar complexity.

## 4. VERBAL PROTOCOL ANALYSIS

After the experiment had concluded, two independent coders, both experienced COBOL programmers, used a coding scheme developed in advance of the experiment [25] to identify the comprehension processes employed by participants. We describe the coding scheme and coding process below. For a more complete description of the coding scheme, please refer to [25].

```
...

WORKING-STORAGE SECTION.

01 ERRORS.
  03 H791            PIC X(4)  VALUE 'H791'.
  03 I086            PIC X(4)  VALUE 'I086'.
  03 F035            PIC X(4)  VALUE 'F035'.

01 WSAA-PROG         PIC X(05) VALUE 'B5361'.
01 WSAA-VERSION      PIC X(02) VALUE '01'.

...
```

Figure 2. Code excerpt from the insurance domain.

## 4.1. The coding scheme

### *4.1.1. Identifying expectation-based hypotheses*

An expectation-based episode can be considered a cycle of stating the expectation, scanning the code for the expectation, and validating or repudiating the existence of the expectation. To identify this process in participants' utterances, coders must look for instances where a pre-generated expectation is stated, where a pre-generated expectation is validated, or a pre-generated expectation is repudiated.

*Expectation*

To be classified as an expectation, the programmer must anticipate the existence of certain goals or structures in the program. A participant uttering phrases such as, 'I *expect* ... *the*' implies a pre-generated expectation. Both the 'expect' and the definitive article ('the') in the utterance, give the impression that there is something predefined in the participants mind. Likewise, by asking '*How*...?' something might be achieved in the system, participants seem to be using a pre-generated expectation [10]. An example from this experiment, when referring to the code in Figure 2, was a programmer's statement: 'I $\underline{\text{expect}}$ to see $\underline{\text{the}}$ normal 01 errors and 01 levels...'.

*Validation*

Validation of an expectation occurs when a programmer locates code consistent with an expectation. If the participant utters something such as '*there* is *the*...', it suggests validation of a pre-generated hypothesis. The 'there' shows that the programmer has found an element of interest. This can be replaced with any phrase that suggests the programmer has found something in the code. The pre-generated nature of the expectation is again suggested by the usage of the definite article: '*the*'. An example from the experiment, using the code in Figure 2 was '...$\underline{\text{it's got}}$ all $\underline{\text{the}}$ normal errors'.

```
...
IF  WSAA-T6654-IX > WSAA-T6654-SIZE
    MOVE H791                TO SYSR-STATUZ
    MOVE T6654               TO SYSR-PARAMS
    PERFORM 600-FATAL-ERROR
END-IF.
...
```

Figure 3. Code excerpt from the insurance domain.

*Repudiation*

Repudiation occurs when a programmer is unable to validate an expectation. If the participant utters a statement such as '*it's not there*...', it may signify that they have as yet, been unsuccessful in validating their pre-generated hypothesis. An (artificial) example using the code in Figure 2 would be, 'the part that deals with the setting up of error codes is not there'.

### 4.1.2. *Identifying inference-based hypotheses*

An inference-based episode can be considered a cycle of scanning the code, inferring an abstracted hypothesis from a beacon in the code, and validating that hypothesis. Thus the programmer goes from scanning to goal identification without a full build-up of knowledge.

*Inferences*

To be classified as an inference, utterances should show the programmer locating code segments that have meaning, using beacons. Utterances that suggest the programmer has just found a beacon and is using it to prompt a hypothesis are composed of three elements:

- identification of the goal associated with the beacon;
- the indefinite article, suggesting that the programmer had no pre-expectation of the goal;
- an initially low level of confidence in their identification of that goal (as the programmer is basing their assertion on a beacon, rather than a full build-up of knowledge).

Using the code in Figure 3, an example from the experiment was, 'there's a table I think, yea, T6654 seems to be a table'. The 'there's', indicates that the programmer is mentally pointing at the code. The 'table' defines the goal. The indefinite article ('a') indicates that this was an unexpected goal, and the 'I think' indicates the programmers level of uncertainty regarding this hypothesis.

*Validation*

Validation of an inference occurs when the programmer inspects the code and confirms the initial meaning attributed to the beacon is consistent with the program. This is signalled by the participant,

```
...
2000-MAINLINE.
     SKIP2
     IF IP-EMP-POL-NO EQUAL TO SA-EMP-POL-NO
     THEN
       NEXT SENTENCE
     ELSE
       PERFORM 8000-CHANGE-OF-POL
              THRU 8000-EXIT
       MOVE SW-OFF TO SW-IS-RECORD-AN-ORPHAN.
     SKIP1
...
```

Figure 4. Code excerpt from the healthcare domain.

again using the indefinite article, along with mentally pointing at the code, but this time, with a higher level of certainty. As in the previous example the validation was, 'Yea, T6654 seems to be a table'. The 'yea' indicates that the programmer has confirmed the hypothesis. The indefinite article suggests that the hypothesis formed was not pre-expected by the programmer.

### *4.1.3. Identifying bottom-up comprehension*

Bottom-up comprehension can be interpreted as a line-by-line analysis of code leading to the eventual deduction of a plan. The programmer assesses lines of source code and, from several lines, deduces a 'chunk' or higher-level abstraction. These abstractions initially represent low-level programming plans. The programmers then aggregate these plans into higher-level plans, which have increasingly domain-orientated goals.

Meaning, at any level, is accessed only after processing at previous (i.e., lower) levels has been completed. This process is uniquely characterized by a full build-up of knowledge and there should be a fairly high level of certainty in the conclusion. A bottom-up episode would typically consist of initial detailed study, aggregation of the detail and the subsequent derivation of a conclusion.

Using the code in Figure 4, an example of bottom-up processing observed during the experiment was: 'Okay, so 2000-MAINLINE. If IP-POL-NUMBER is equal to SA-POL-NUMBER then NEXT SENTENCE else perform 8000-CHANGE-OF-POL... Okay so these are performed while they are equal'.

## 4.2. The coding process

Two independent Coders (talk-aloud interpreters), using our extended version of Shaft's coding scheme [29], were asked to distinguish between these three types of comprehension process. The coding scheme contained detailed definitions of expectation-based processing, inference-based processing, bottom-up processing, and examples of each. It also defined rules for assigning utterances

Table III. Consistency in classification between the two Coders: $\kappa = 0.984$.

| | Coder 1 | | | |
|---|---|---|---|---|
| Coder 2 | Expectations | Inference-based | Bottom-up | Not coded |
| Expectations | 22 | 0 | 0 | 2 |
| Inference-based | 0 | 67 | 0 | 0 |
| Bottom-up | 0 | 0 | 24 | 1 |
| Not coded | 0 | 2 | 0 | 210 |

to each of these processes. In line with Good's recommendations for analysis schemas [16], a decision tree was provided for the Coders analysing the talk-aloud transcripts.

Any disagreements between the Coders, after they had independently translated the programmers' utterances, were discussed in a meeting. Only the Coders were present at this meeting and the experiment designers were not allowed any input into their discussions. As a result of the meeting, Coders were allowed to review their analysis. The analysed results were then given to the experiment designers to interpret for consistency. Over 95% of their utterance classifications were consistent.

In cases where consistency between two observers is to be assessed, Hartmann [30] concluded that the Kappa test should be employed. This test, determines the level of agreement between two sets of observations based on the number of observation that are consistent and the number that are inconsistent. The results from this test, presented in Table III, suggest that the Coders were in 'almost perfect' agreement [31] as Kappa exceeded 0.8. This suggests that coders were able to distinguish reliably between expectations, inferences, and bottom-up comprehension processes. Furthermore, examination of Table III indicates that coders never disagreed between comprehension categories (expectation, inference and bottom-up). This high level of agreement, as shown in this table, indicates that the Coder's manual was a reliable classification instrument for distinguishing between these comprehension processes.

There were three reasons why certain phrases were categorized as 'not coded', forming what Good [16] refers to as a 'bucket category'. Firstly, when phrases were simply translations of program statements from COBOL to English, secondly, when phrases were just programmer summarizes of their understanding of a program statement, and finally, any participant/experimenter interaction.

## 5. RESULTS, ANALYSIS, AND DISCUSSION

### 5.1. Hypothesis 1

To assess Hypothesis 1, we must identify whether expectation-based and inference-based processing are distinct elements of comprehension. Coders were able to reliably distinguish between these types of processes using the Coder's manual, and found instances of both in the programmers' utterances.

Table IV. Binomial test for the occurrence of inference-based comprehension.

| | Category | $N$ | Observed proportion | Test proportion | Asymp. Sig. (one-tailed) |
|---|---|---|---|---|---|
| Group 1 | Not inferences | 49 | 0.42 | 0.33 | 0.22[a] |
| Group 2 | Inferences | 67 | 0.58 | | |
| Total | | 116 | 1.00 | | |

[a] Based on $Z$ approximation.

Table V. Insurance programmers: reconciled results for a familiar application domain.

| Participant | Expectations | Inferences | Bottom-up |
|---|---|---|---|
| P1 | 2 | 7 | 2 |
| P2 | 3 | 3 | 1 |
| P3 | 1 | 2 | 1 |
| P4 | 6 | 8 | 3 |
| Totals | 12 | 20 | 7 |

These results indicate support for Hypothesis 1; that programmers use both expectation and inference based (as well as bottom-up) comprehension processes (see Table III).

Of the episodes identified, approximately 20% were expectation-based, 60% were inference-based and 20% were bottom-up. These results suggest that the programmers relied heavily on inference-based comprehension during their sessions, a suggestion supported by a binomial test of the occurrence of inference-based comprehension processes. Table IV shows that the proportion of inference-based comprehension observed is significantly different from 0.33, the proportion that would be expected if all three comprehension processes were employed equally.

## 5.2. Hypotheses 2(a), 2(b), and 2(c)

To examine Hypotheses 2(a), 2(b), and 2(c), the comprehension processes employed by the programmers are analysed with respect to application domain familiarity. This breakdown of results is presented in Tables V and VI for insurance programmers and Tables VII and VIII for healthcare programmers.

We relied upon SAS PROC GLM to analyze Hypotheses 2(a), 2(b), and 2(c). For all analyses, we include both familiarity with the application domain, the application domain of the computer program and their interactions as factors in the model. Because our data was distributed in a non-normal

Table VI. Insurance programmers: reconciled results for an unfamiliar application domain.

| Participant | Expectations | Inferences | Bottom-up |
|---|---|---|---|
| P1 | 0 | 3 | 1 |
| P2 | 0 | 3 | 0 |
| P3 | 0 | 4 | 0 |
| P4 | 2 | 3 | 2 |
| Totals | 2 | 13 | 3 |

Table VII. Healthcare programmers: reconciled results for a familiar application domain.

| Participant | Expectations | Inferences | Bottom-up |
|---|---|---|---|
| S5 | 1 | 3 | 3 |
| S6 | 3 | 0 | 1 |
| S7 | 3 | 5 | 6 |
| S8 | 1 | 3 | 3 |
| Totals | 8 | 11 | 13 |

Table VIII. Healthcare programmers: reconciled results for an unfamiliar application domain.

| Participant | Expectations | Inferences | Bottom-up |
|---|---|---|---|
| P5 | 0 | 7 | 0 |
| P6 | 0 | 8 | 0 |
| P7 | 0 | 5 | 1 |
| P8 | 0 | 3 | 0 |
| Totals | 0 | 23 | 1 |

fashion, we followed the approach recommended by Conover [32] to address concerns about meeting the assumptions of a parametric test. Specifically, we converted the data to ranks and replicated each analysis. When the analysis of the ranks confirms the parametric analysis, one can conclude that the departure from normality is not enough to violate the assumptions of the model and the results of the parametric (i.e., GLM) analysis are valid [32]. The analysis for each specific hypothesis is presented below.

Table IX. Analysis of Hypothesis 2(a): number of expectations.

| Source | DF | Type III SS | Mean square | $F$ value | $Pr > F$ |
|---|---|---|---|---|---|
| Familiarity | 1 | 20.25 | 20.25 | 11.57 | 0.005 |
| Application domain of computer program | 1 | 0.25 | 0.25 | 0.14 | 0.712 |
| Familiarity × application domain of program | 1 | 2.25 | 2.25 | 1.29 | 0.279 |

### *5.2.1. Hypothesis 2(a): expectation-based comprehension*

Hypothesis 2(a) states that when a programmer is familiar with the application domain, they will use more expectation-based processing than when unfamiliar with the application domain. To test this hypothesis, the number of expectations stated by each programmer in each application domain serves as the dependent variable. The model is significant ($F_{3,12} = 4.33$, $p$-value $= 0.03$, $r^2 = 0.52$). The main effect of familiarity with the application domain was the only significant factor ($F_{1,12} = 11.57$, $p$-value $= 0.005$).

As described above, we replicated the analysis after converting our data to ranks. The results confirmed the results obtained from the parametric (GLM) analysis; the model yields a significant fit and the only significant effect is that of familiarity with the application domain. Our findings (see Table IX) support Hypothesis 2(a). When programmers were familiar with the application domain, they stated more expectations than when they were unfamiliar with the application domain. Further, we did not detect a significant difference in the number of expectations based on the application domain of the program, nor any interaction between familiarity and application domain of the program being studied.

### *5.2.2. Hypothesis 2(b): inference-based comprehension*

Hypothesis 2(b) states that, when a programmer is familiar with the application domain, they will use more inference-based processing than when unfamiliar with the application domain. To test this hypothesis, the number of inferences stated in each application domain serves as the dependent variable. The fit for this model was not significant ($F_{3,12} = 1.78$, $p$-value $= 0.20$, $r^2 = 0.31$). However, the main effect for application domain of the computer program was significant ($F_{1,12} = 4.99$, $p$-value $= 0.05$). Therefore, we repeated the analysis including only application domain of the computer program as a factor in the model. This analysis was significant ($F_{1,15} = 5.65$, $p$-value $= 0.03$, $r^2 = 0.29$); the $F$- and $p$-values are attributed to the influence of the main effect of the application domain of the computer program. Similar to the approach used in testing Hypothesis 2(a), we converted the data to ranks and obtained results similar to those detected in the parametric analysis. The analysis of the full model does not yield a significant fit. The analysis including only the application domain of the computer program was nearly significant ($F_{1,12} = 3.36$, $p$-value $= 0.08$). Our results indicate that the number of inferences stated by programmers is best explained by the application domain of the computer program.

Table X. Number of inferences stated by each programmer in each application domain.

| | | Number of inferences | | |
|---|---|---|---|---|
| | Participant | Insurance application domain | Healthcare application domain | Overall means for programmer group by application domain familiarity |
| Insurance programmers | 1 | 7 | 3 | |
| | 2 | 3 | 3 | |
| | 3 | 2 | 4 | |
| | 4 | 8 | 3 | 4.13 |
| Healthcare programmers | 5 | 7 | 3 | |
| | 6 | 8 | 0 | |
| | 7 | 5 | 5 | |
| | 8 | 3 | 3 | 4.25 |
| Mean: by application domain | | 5.38 | 3.00 | 4.19 |

Table XI. Analysis of Hypothesis 2(c): number of bottom-up statements.

| Source | DF | Type III SS | Mean square | $F$ value | $Pr > F$ |
|---|---|---|---|---|---|
| Familiarity | 1 | 16.00 | 16.00 | 10.11 | 0.008 |
| Application domain of computer program | 1 | 4.00 | 4.00 | 2.53 | 0.138 |
| Familiarity $\times$ application domain of program | 1 | 1.00 | 1.00 | 0.63 | 0.442 |

Clearly, Hypothesis 2(b) is not supported. To examine the data more closely, we display the data and marginal means in Table X. One can see that programmers uttered more inferences in the insurance domain than the healthcare domain regardless of familiarity with the application domain. It appears that application domains may facilitate or inhibit the generation of inferences. However, this finding must be treated as tentative because the full model (including both factors in our experimental design) did not yield a significant fit to the data.

### *5.2.3. Hypothesis 2(c): bottom-up comprehension*

Hypothesis 2(c) states that when a programmer is unfamiliar with the application domain, they will rely on bottom-up comprehension and use this process more than when they are familiar with the application domain. To test this hypothesis, the number of bottom-up statements made by each programmer in each application domain serves as the dependent variable. The model is significant ($F_{3,12} = 4.42$, $p$-value $= 0.03$, $r^2 = 0.53$). The main effect of familiarity with the application domain (see Table XI) was the only significant factor ($F_{1,12} = 10.11$, $p$-value $= 0.008$).

Table XII. Bottom-up episodes by quartile.

| Quartile | Number of bottom-up episodes in familiar application domain |
|---|---|
| 1st | 3 |
| 2nd | 2 |
| 3rd | 7 |
| 4th | 8 |

Similar to our approach for testing Hypotheses 2(a) and 2(b), we converted our data to ranks and repeated our analysis. The analysis of the ranks of the data, confirm the findings from our parametric analysis (the model yields a significant fit and the only significant effect is that of familiarity with the application domain), indicating that the departure from normality was not enough to violate the assumptions of the original analysis [32]. However, the direction of the relationship is at odds with Hypothesis 2(c). The number of bottom-up statements was greater when the participant is familiar with the application domain of the computer program. We did not detect significant differences based on the application domain of the computer program, nor did we detect an interaction between familiarity and the application domain of the computer program.

## 6. STUDY CONCLUSIONS

As expected, programmers' familiarity with the application domain was associated with a comprehension process that relied upon pre-generated expectations. Using the knowledge of the application domain, programmers were able to pre-generate expectations about typical program goals from that application domain.

It was hypothesized that when familiar with the application domain, programmers would be more likely to infer program plans than when they were not. However, our results indicate that inference based comprehension was not related to application domain knowledge. Our tentative finding from Section 5.2.1 suggests that programmers studying the insurance program performed more inference-type comprehension processes than those studying the healthcare program, regardless of application domain familiarity. Thus, the programmers' behaviour would seem to be affected by some independent variable in the program that we have not yet identified. This might prove to be an interesting avenue for future work.

As regards the surprising prevalence of bottom-up processing by programmers familiar with the application domain, we suggest one possibility: when familiar with the application domain, programmers had sufficient time to determine the program's overall meaning and subsequently may have worked through the code in a detailed, line-by-line fashion (after they had obtained this overview). To investigate this possibility further, we examined the number of bottom-up utterances carried out by programmers in familiar application domains by time quartile. This analysis shows that most bottom-up episodes performed in familiar application domains, occurred towards the end of the comprehension session (see Table XII). This supports the idea that when a programmer is familiar with the application

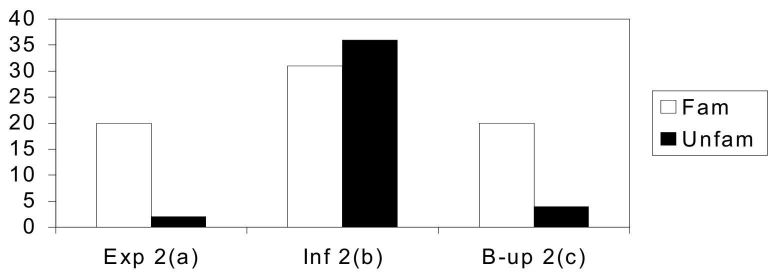

Figure 5. Participants' overall usage of variants with respect to application domain familiarity.

domain they revisit the system to confirm in detail what it was they learned in overview earlier. The use of a more efficient process in the early stages of comprehension may allow (application domain-familiar) programmers more time to revisit the code and gain a deeper understanding of program details in the given time.

In summary, from the results obtained during this experiment, it was clear that expert programmers employ all three types of comprehension processes and that familiarity led to increased use of pre-generated expectations. Figure 5 graphically depicts the total number of expectations, inferences, and bottom-up processing, used by programmers for both the familiar and unfamiliar application domains during the experiment.

## 7. STRENGTHS AND LIMITATIONS OF THE STUDY

The major strengths of this study are its use of professional (industrial) programmers, the formal, open (and thus repeatable) coding of verbal protocols, the use of real medium-sized [18] code from the two industrial domains, and the fully crossed design. Many studies in the area of software comprehension use students as participants in empirical studies. Indeed, this would be justified if this study examined novice programmers, but this research focuses on actual practice and programmers' familiarity with real-world domains. So it was imperative that professional, experienced programmers served as participants. The use of these programmers made the results more justifiably sound and valid.

Another strength of the study was its usage of code from the participants' industrial setting. Rather than the experiment designer writing 'typical' code from these application domains, it was thought to be more accurate if an independent programmer from each domain wrote the code, which in both cases was just under 1200 lines.

The third strength of this experiment was that all verbal protocols were coded by two independent coders, both having a working knowledge of the COBOL programming language. Coders worked independently of each other as well as the designer, and only used the Coder's Manual to complete the task. Reliability was checked and the results from both coders showed over 98% agreement.

This highlights the adequacy of the process descriptions given in the Coders Manual, formalizing these subtle comprehension processes and making them differentiable.

A final strength of this study is that the effect of application domain familiarity was fully crossed in this study. All programmers worked in both a familiar and an unfamiliar application domain, similar to previous investigations into the impact of application domain knowledge [15,33]. However, in this study, the effect of application domain knowledge was more carefully tested because, for half of the participants, the insurance domain was familiar (and healthcare unfamiliar). For the other participants this was reversed: healthcare was familiar and insurance unfamiliar.

Potential limitations of this study included the small sample size and the small size of the programs used. This study used eight programmers, four from each application domain. Future work should increase this sample size, using different cohorts to assess how representative the study was. Also, although each program used in this experiment was larger than the majority of prior studies, current systems often have millions of lines of code. Code of this scale was chosen to keep participation time within 1.5 hours, and thus make participation more attractive to professional programmers.

Another limitation of this work was the fact that the source code used in the experiment was in hardcopy format. This was mainly due to the security concerns of both companies. However, many empirical studies of programmers present participants with hardcopy versions of the source code [3,4,15,19]. Using a hardcopy representation, participants are unable to execute the code or use searching tools—a facility used prevalently by software engineers [34]. Observational studies, such as those carried out by Von Mayrhauser *et al.* [35], allow programmers to use tools and execute the programs; however, the extent to which these facilities assisted the comprehension process, was not assessed. In addition, note that in some studies where programmers relied upon an on-line tool, the programmers were all from the same firm and were observed in their work setting [35]. However, our study relied on programmers from multiple firms. Therefore, to provide access to an on-line tool in an experimental setting creates a potential confound. Minimally, it would be necessary to have additional experimental controls to ensure that all programmers had equal facility with the tool prior to conducting the experiment (e.g., control for experience with the tool and/or provide training in using the on-line tool such that programmers were equally competent). Future work should assess the role that soft representations of the system, and indeed the executing system, can play in software comprehension processes.

A final limitation of the study was the task given to the participants. Very seldom, it is envisaged, do programmers sit down with code and understand code for its own sake. Typically the comprehension they undertake is shaped by a wider task such as debugging [7,16,36]. As the programmers were asked to generate a summary of the programs in this study, this could be considered a form of re-documentation; the lack of a more grounded task context may lower the external validity of the study [37].

## 8. CONCLUSION

This paper has described an experiment that investigated the comprehension processes employed by programmers when studying software. Talk-aloud utterances by programmers were categorized in a manner similar to that carried out in [29]. However, this work extended the original categorization, identifying three distinct types of comprehension, namely:

- expectation-based comprehension, where the programmer has pre-generated expectations of the code's meaning;
- inference-based comprehension, where the programmer derives meaning from clichéd implementations in the code;
- bottom-up comprehension, where the programmer works through the code on a line-by-line basis, building up an understanding methodically.

From the experiment, it was discovered that expert programmers employ all three strategies during program comprehension. However, if the application domain is familiar, programmers rely on pre-generated expectations more than when they are unfamiliar with the application domain. This is important as an expectation-based comprehension process is considered to be more powerful than other approaches to software comprehension [14]. In contrast, programmers unfamiliar with the application domain rarely relied on expectations during comprehension. When programmers lack knowledge of the application domain, it appears that they are unable to pre-generate expectations about the code's meaning.

Programmers also used inferences extensively, possibly due to their language expertise and their reluctance to perform bottom-up comprehension, which may be cognitively more demanding than either expectation- or inference-based processes.

Future work should look at the interplay between pre-generated expectations, inferences, and bottom-up comprehension. This work should examine the factors that cause programmer to switch from one comprehension process to another, or to emphasize one over another.

Also, this research suggests that inference-based processing, where programmers use beacons in the code to detect its functionality, is a prevalent software comprehension technique. Approximately 60% of all the comprehension processing observed in this study is of this type. If this finding is buttressed by other studies, further research should try to characterize beacons' usefulness and use by programmers, in the vein of [38,39]. This would seem like a fruitful avenue for defining coding conventions that facilitate comprehension.

Finally, future work should seek an explanation for the usage of bottom-up episodes by experienced programmers during comprehension of familiar software systems. Bottom-up comprehension has been associated with 'degenerative' behaviour of programmers unfamiliar with the application domain [14]. One plausible explanation for its prevalence in this experiment is that these programmers go through the program a second time, methodically, getting a deeper understanding of the program's meaning (see Table XI). However, this explanation is provisional and should be confirmed by future research.

**ACKNOWLEDGEMENTS**

The authors wish to thank Professor Thomas Green (University of Leeds), Dr Chris Douce (Feedback Instruments, U.K.), Dr Jean Saunders, and the Software Visualisation and Cognition Research Group (University of Limerick), for valuable suggestions and constructive criticism received on earlier drafts of this paper. We also wish to thank the anonymous reviewers for their constructive comments and for their willingness to engage while we undertook these changes. This work was undertaken as part of the EI Basic Research Project: SC\1999\035\Y and as part of the Science Foundation Ireland Investigator Programme, B4-STEP.

## REFERENCES

1. De Lucia A, Fasolino AR, Munro M. Understanding function behaviours through program slicing. *Proceedings 4th Workshop on Program Comprehension*. IEEE Computer Society Press: Los Alamitos CA, 1996; 9–19.
2. Rajlich V. Program reading and comprehension. *Proceedings of Summer School on Engineering of Existing Software*, Dipartimento di Informatica, University of Bari, Italy, 1994; 161–178.
3. Shneiderman B, Mayer R. Syntactic/semantic interactions in programmer behaviour. *International Journal of Computer and Information Sciences* 1979; **8**(3):10–24.
4. Pennington N. Comprehension strategies in programming. *Empirical Studies of Programmers: Proceedings of the 2nd Workshop*. Ablex: Norwood NJ, 1987; 100–113.
5. Linger RC, Mills HD, Witt BI. *Structured Programming: Theory and Practice*. Addison-Wesley: Boston MA, 1979; 402.
6. Eysenck MW, Keane MT. *Cognitive Psychology: A Student's Handbook*. Psychology Press: East Sussex, U.K., 2000; 151–182.
7. Detienne F. *Software Design—Cognitive Aspects*. Springer: London, 2002; 75–102.
8. Ramalingham V, Weidenbeck S. An empirical study of novice program comprehension in the imperative and object-oriented styles. *Proceedings of 7th Workshop on Empirical Studies of Programmers*. Ablex: Norwood NJ, 1997; 124–139.
9. Green TRG. Personal correspondence, 2002.
10. Letovsky S. Cognitive processes in program comprehension. *Empirical Studies of Programmers: Proceedings of the 1st Workshop*. Ablex: Norwood NJ, 1986; 58–79.
11. Von Mayrhauser A, Vans AM. From code understanding needs to reverse engineering tool capabilities. *Proceedings 6th International Workshop on Computer-Aided Software Engineering (CASE '93)*. IEEE Computer Society Press: Los Alamitos CA, 1993; 230–239.
12. Von Mayrhauser A, Vans AM. Program understanding behavior during debugging of large scale software. *Empirical Studies of Programmers: Proceedings of the 7th Workshop*. Ablex: Norwood NJ, 1997; 157–179.
13. Blum BI. Volume, distance, and productivity. *Journal of Systems and Software* 1989; **10**(3):217–226.
14. Brooks R. Towards a theory of the comprehension of computer programs. *International Journal of Man–Machine Studies* 1983; **18**(6):543–554.
15. Shaft TM, Vessey I. The relevance of application domain knowledge: The case of computer program comprehension. *Information Systems Research* 1995; **6**(3):286–299.
16. Good J. Programming paradigms, information types and graphical representations: Empirical investigations of novice program comprehension. *Doctoral Dissertation*, University of Edinburgh, Edinburgh, Scotland, 1999; 19–207.
17. Von Mayrhauser A, Vans A, Howe AE. Program understanding behaviour during enhancement of large scale software. *Journal of Software Maintenance: Research and Practice* 1997; **9**(5):299–327.
18. Von Mayrhauser A, Vans AM. Program understanding: Models and experiments. *Advances in Computers* 1995; **40**(4):25–46.
19. Soloway E, Ehrlich K. Empirical studies of programming knowledge. *IEEE Transactions on Software Engineering* 1984; **10**(5):595–609.
20. O'Brien MP, Buckley J. Inference-based and expectation-based processing in program comprehension. *Proceedings 9th International Workshop on Program Comprehension*. IEEE Computer Society Press: Los Alamitos CA, 2001; 71–79.
21. Wiedenbeck S. Beacons in computer program comprehension. *International Journal of Man–Machine Studies* 1986; **25**(6):697–709.
22. Storey M-AD, Fracchia FD, Müller HA. Cognitive design elements to support the construction of a mental model during software exploration. *The Journal of Systems and Software* 1999; **44**(1):171–185.
23. Tilley S, Paul S, Smith D. Towards a framework for program understanding. *Proceedings 4th International Workshop on Program Comprehension*. IEEE Computer Society Press: Los Alamitos CA, 1996; 19–28.
24. Rist R. Plans in programming: definition, demonstration, and development. *Proceedings of the 1st Workshop on the Empirical Studies of Programmers*. Ablex: Norwood NJ, 1986; 28–47.
25. O'Brien MP, Buckley J. The GIB talk-aloud classification schema. *Technical Report 2000-1-IT*, Limerick Institute of Technology, 2000. Available on request from authors.
26. Ericsson K, Simmons H. Verbal reports as data. *Psychological Review* 1980; **89**(3):215–251.
27. Russo J, Johnson E, Stephens D. The validity of verbal protocols. *Memory and Cognition* 1989; **17**(6):759–769.
28. Lind R, Vairavan K. An experimental study of software metrics and their relationship to software development effort. *IEEE Transactions on Software Engineering* 1989; **15**(5):649–653.
29. Shaft TM. The role of application domain knowledge in computer program comprehension and enhancement. *Doctoral Dissertation*, Pennsylvania State University, University Park PA, 1992; 213–269.
30. Hartmann D. Considerations in the choice of inter-observer reliability estimates. *Journal of Applied Behaviour Analysis* 1977; **10**:103–116.
31. Landis JR, Koch GG. The measurement of observer agreement for categorical data. *Biometrics* 1977; **33**(1):159–174.

32. Conover WJ. *Practical Nonparametric Statistics* (3rd edn). Wiley: New York NY, 1999; 584.
33. Shaft TM, Vessey I. The relevance of application domain knowledge: Characterizing the computer program comprehension process. *The Journal of Management Information Systems* 1998; **15**(1):51–78.
34. Singer J, Lethbridge T. Practices of software maintenance. *Proceedings International Conference on Software Maintenance*. IEEE Computer Society Press: Los Alamitos CA, 1998; 139–145.
35. Von Mayrhauser A, Vans M, Lang S. Program comprehension and enhancement of software. *Proceedings of the 15th IFIP World Computing Congress—Information Technology and Knowledge Systems*. Austrian Computer Society: Vienna, Austria.
36. Littman D, Pinto J, Letovsky S, Soloway E. Mental models and software maintenance. *Empirical Studies of Programmers: Proceedings of the 1st Workshop*. Ablex: Norwood NJ, 1986; 80–98.
37. Perry D, Porter A, Votta L. A primer on empirical studies. *Proceedings International Conference on Software Maintenance*. IEEE Computer Society Press: Los Alamitos CA, 1997; 657–658.
38. Gellenbeck EM, Cook CR. An investigation of procedure and variable names as beacons during code comprehension. *Empirical Studies of Programmers: Proceedings of the 4th Workshop*. Ablex: Norwood NJ, 1991; 65–81.
39. Gellenbeck EM, Cook CR. Does signalling help professional programmers read and understand computer programs. *Empirical Studies of Programmers: Proceedings of the 4th Workshop*. Ablex: Norwood NJ, 1991; 82–98.

## AUTHORS' BIOGRAPHIES

**Michael P. O'Brien** is currently completing a PhD in Computer Science at the University of Limerick, Ireland, and holds both an honours BSc degree in Information Systems and an MSc degree in Information Technology. His research interests include cognitive psychology, software comprehension strategies, empirical studies of programmers and software evolution. He has published actively at international conferences, workshops and seminars in this domain, and won the Dwyer Memorial Award for best paper at a national science and technology research colloquium, for an earlier phase of this research.

**Jim Buckley** obtained an honours BSc degree in Biochemistry from the University of Galway in 1989. In 1994 he was awarded an MSc degree in Computer Science from the University of Limerick and he followed this with a PhD in Computer Science from the same University in 2002. He currently works as a lecturer in the Computer Science and Information Systems Department at the University of Limerick, Ireland. His main research interest is in theories of software comprehension, software reengineering and software maintenance. In this context, he has published actively at many peer-reviewed conferences and workshops.

**Teresa M. Shaft** is Assistant Professor of MIS, Price College of Business, University of Oklahoma. She studies the cognitive processes of information systems professionals during software development and maintenance and the use of information systems in environmental management. She has published research in *Information Systems Research, Journal of MIS, Journal of Industrial Ecology*, etc. She is a co-founder of IS-CORE, a special interest group of AIS. Her research has been supported by the National Science Foundation, U.S.A.