Inference-based and Expectation-based Processing in Program Comprehension

Michael P. O'Brien, Jim Buckley
Department of Information Technology
Limerick Institute of Technology
michael.obrien@lit.ie jim.buckley@lit.ie

Abstract

This paper formally distinguishes between two variants of top-down comprehension (as originally described by Brooks and Soloway). The first is 'inference-based' comprehension, where the programmer derives meaning from clichéd implementations in the code. The second is 'expectation-based' comprehension, where the programmer has pre-generated expectations of the code's meaning.

The paper describes the distinguishing features of the two variants, and uses these characteristics as the basis for an empirical study. This study establishes their existence, and identifies their relationship with programmers' domain and coding standards familiarity.

1. Introduction.

Program comprehension is a major part of software development especially at the maintenance phase. Studies have shown that up to 70% of lifecycle costs are consumed within the maintenance phase, with 50% of these costs relating to comprehension alone [1], [2]. Hence up to 35% of the total lifecycle costs can be directly associated with simply understanding the code. Despite being such a significant activity, research in the area of program comprehension is relatively new. Only in the past few decades have researchers attempted to evaluate the factors that affect comprehension and develop processes that aim to describe how programmers comprehend or understand code [13].

Current research suggests that programmers attempt to understand code using two main strategies, namely top-down and bottom-up comprehension. Top-down comprehension is where the programmer utilises knowledge about the domain of the program, and maps it to the actual code itself. Bottom-up comprehension then on the other hand, is where understanding is built from the bottom-up, by reading the code and then mentally chunking or grouping these statements into higher-level abstractions. However it is unlikely that programmers rely on either one of these strategies exclusively. Instead, the literature suggests, they subconsciously choose one of these to be their pre-dominant strategy, based on their knowledge of the domain under study. [3], [4].

1.1 Background.

Several theories of programmers' comprehension processes have been proposed [5], [6], [7], [8], [9]. This paper specifically distinguishes between the two defined by Brooks and Soloway et al, which in the past, have been grouped together. Brooks presents a theory of top-down comprehension process by which programmers' pre-generate hypotheses. These hypotheses may come from other programmers, expertise, or documentation. Hypotheses may be subsequently subdivided into finer hypotheses until they can be bound to particular code segments (see Fig.1).

The starting point for the theory is an analysis of the structure of the knowledge required when a program is comprehended. This theory views knowledge as being organised into distinct domains that bridge between the original problem and the final program. The program comprehension process is one of reconstructing knowledge links between these domains for the programmer [6]. This reconstruction process is theorised to be top-down and hypothesis driven where

Proceedings of the 9th International Workshop on Program Comprehension (IWPC'01)
1092-8138/01 $10.00 © 2001 **IEEE**

an initially vague and general hypothesis is refined and elaborated upon based on information extracted from the program text and other relevant documentation. However Brooks did not carry out any formal empirical evaluation of this theory, although Von Mayrhauser et al have since performed empirical studies that support it [5], [12].

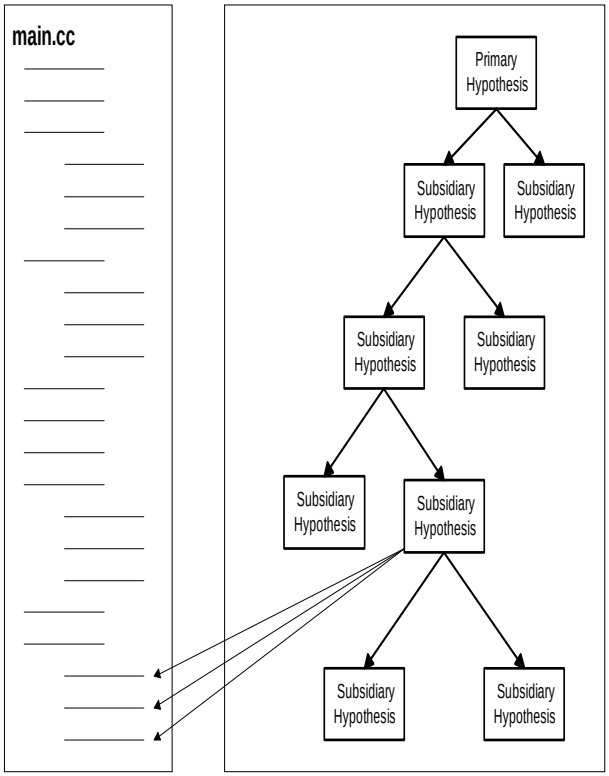

Fig. 1 – Brooks' model of program comprehension

Soloway's work, which is often taken as top-down exclusively [15], [5], [16], has elements of bottom-up processing. Soloway hypothesises that programmers initially scan the code using 'shallow reasoning', for critical line(s) or beacons, which they use to prompt plans in their mental model. Then, using top-down comprehension, programmers seek to confirm or validate the plan they have generated from the code. From Soloway's empirical validation there was no evidence that pre-expectations were employed; as subjects had no knowledge of the code they were about to see. However, Soloway didn't set out to study these so-called 'pre-expectations' specifically.

Essentially then, the differing characteristic between the two, *is what triggers plans in programmers' minds when understanding systems.* Brooks suggests that the trigger is pre-generated hypotheses, which are matched up with the code. Soloway's studies suggest that the trigger is beacons in the code, which result in the generation of plans, which subsequently, lead to validation.

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 a new, unexpected plan.

1.2 Paper Structure.

This paper aims to differentiate between, and establish, the co-existence of the processes suggested by Brooks and Soloway. It does this by carrying out experiments where industrial software engineers try to understand software from a familiar and unfamiliar domain. Programmers' talk-aloud is captured and categorised into episodes of expectation-based, inference-based, and bottom-up comprehension by independent coders, in line with a coding schema developed at the Limerick Institute of Technology [14]. This coding scheme aims to distinguish between expectation-based, inference-based, and bottom-up comprehension, so that independent coders can reach a high level of independent agreement. This level of agreement is seen as validation for the differentiation between expectation-based and inference-based comprehension.

This paper subsequently aims to establish the relationship between domain familiarity and predominance of strategy. The familiar domain, it is hypothesised, will allow the programmer to generate more plan expectations and thus follow Brooks top-down comprehension process. It is also likely that organisational experience will play a role here as structural expectations may occur, based on programmers' expertise of organisational coding standards. The unfamiliar domain, on the other hand, will force programmers to rely more on bottom-up processing.

2. Experimental Study.

2.1 Hypothesis.

This research attempts to illustrate the existence of expectations and inferences in programmers' comprehension sessions.

Once established, the research will attempt to examine the effect of domain familiarity on these software comprehension processes. Programmers, who are familiar programming in a particular domain to particular standards, will rely more on pre-generated expectations than those who are unfamiliar with the domain (see Fig. 2). As mentioned above, these programmers may also rely on inferences during program comprehension. This, we suggest, is due to their ability to identify beacons in the code that suggest functionality, and their pre-existing knowledge of the domain functionality.

On the other hand, programmers' unfamiliar programming in a particular domain will have to rely more on bottom-up strategies, than programmers familiar with the domain. 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 as to the meaning of typical code from a domain such as this.

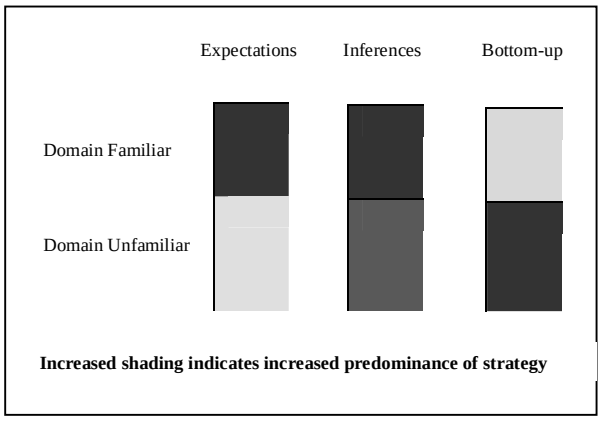

Fig. 2 - Categorisation of Domains with respect to Familiarity

2.2 Methodology.

2.2.1 Subjects. Eight professional/industrial programmers from two independent software companies (4 from a healthcare company and 4 from an insurance company) participated in this experiment. Subjects were employed in developing and maintaining COBOL systems, and were each given £20 to cover any expenses incurred as a result of their participation. Subjects were also given confidential feedback on the comprehension strategies they employed during the comprehension sessions.

Table 1 – Analysis of Subject Questionnaires

AVERAGE	Healthcare Company	Insurance Company
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 Current Company	2.6 years	5 years

From a questionnaire, we determined that the healthcare company programmers had, on average, much more experience in healthcare, and the insurance programmers had much more experience in insurance. Indeed, the insurance company had no experience whatsoever programming healthcare type systems.

2.2.2 Experiment Design. A background questionnaire was administered prior to the study to assess the grades, experience programming in COBOL, and subjects' estimation of their self-knowledge of the COBOL language (see Table 1 - Analysis).

The experiment required subjects to comprehend two COBOL programs, each consisting of just over 1200 lines of code, which were presented in hardcopy format. The subjects were told that, at the end of their

comprehension sessions, they would have to give a verbal summary of the software.

Of the two programs used in the study, the first was from the insurance company and the other from the healthcare company. All programmers were required to study both programs; hence all programmers saw familiar and unfamiliar code. This was important because to fully test our hypothesis, we required programmers to comprehend programs from both familiar and unfamiliar domains.

All subjects were requested to think aloud for the duration of the experiment so that their comprehension strategies could be captured. If at any stage they begin to work silently, they were simply prompted "What are you thinking now?" by the experimenter.

Two independent coders, both experienced in COBOL, used a coding scheme developed in advance of the experiment, to identify the processes employed by subjects, from the talk-alouds produced by programmers.

2.2.3. Procedure. The experiment itself took place in three segments, a practice session (code from a generic domain), and two experimental sessions (code from each application domain). At the beginning of each segment, all subjects were given a hard copy of the source code (132 Paper Matrix) and asked to study the program so as to gain as complete an understanding of the program as the time allowed, whilst thinking aloud. Programmers were presented with unfamiliar code in one session, and familiar code in the other.

When each session concluded, subjects were given a set of puzzle-like exercises to distract their attention from the task previously undertaken. This gave them a chance to clear their minds, so they could start afresh in the next session without having the opportunity to carryover details from the session just undertaken.

2.2.4 Program Source Code. The insurance program used in the experiment created a new company insurance policy file using data from the input policy and employee files. The new policy file had to contain all relevant policy and employee details, i.e. contained only those policies with current or impending effective periods.

The healthcare program read an employee file and healthcare policy file and generated a file with active healthcare policies. This healthcare policy file was keyed on Social Security Number.

It was vital that the programs from the two domains be comparable on an overall complexity metric. Since many different program complexity measures are highly correlated with source lines of code (SLOC), we used SLOC as one of our complexity standards [10]. We also acquired four professional COBOL programmers to independently assess the complexity of the two programs. They rated the programs on a 1 - 5 basis, where 1 was 'simple' and 5 was 'very difficult'. Results of this rating are shown in Table 2.

Table 2 – Results of Source Code Analysis

COBOL Programmer	Healthcare Program	Insurance Program
1	3	4
2	3	3
3	3	4
4	4	3
AVERAGE >	**3.25**	**3.5**

3. Verbal Data Analysis.

The verbal protocol data (talk-aloud) was analysed to determine the nature of the comprehension processes employed by subjects, in both the familiar and unfamiliar application domains. Two independent coders, using our extended version of Shaft's Coders Manual [11], were asked to distinguish between the types of comprehension process used, namely, expectation-based, inference-based, and bottom-up. The modified Coders Manual contained detailed definitions, examples of pre-generated expectations, inferences, and bottom-up, along with rules for assigning codes. Essentially, this categorisation extends Shaft's categorisation from 'Hypothesis' and 'Inferences', to 'Pre-Generated Hypotheses', 'Inferred Hypotheses', and 'Bottom-up (Shaft refers to bottom-up as inferences)'.

The coders, when analysing the verbal data, attempted to categorise top-down comprehension into two different categories, namely, inference based and expectation-based comprehension. Any disagreements

were reconciled in joint meetings between the two coders.

The following is a sample of some of our modifications to the Coders Manual. This extended manual was available to the two coders for the duration of the coding process.

3.1 Identifying Expectation-based Hypotheses.

In the experiment, a hypothesis is pre-generated (expected) or not, based on the subjects' use of the definite article, i.e., **'the'**, and words 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;
  }
}
```

Fig. 3 – C code implementation of a Bubble Sort

From Fig. 3, examples of expectation-based hypotheses from verbal data protocols include: "I expect to see **the** bubble sort…", or "Where is **the** section that deals with…".

Expectations may also be classified by words, which suggest omission, such as "expect", "should be", and "normally". For example, programmers might say, "there *should be* some sort here", "there is *normally* some sort procedure here….". In essence, this expectation-based episode can be considered a cycle of stating the expectation, scanning the code for the expectation, and validating the existence of the expectation. Sometimes however, the only thing apparent in the protocol may be the validation alone.

3.2 Identifying Inference-based Hypotheses.

Inference-based Hypotheses are identified primarily by the use of the indefinite article in the verbal data, i.e., **'a'**. Examples of inference-based hypotheses from the code in Fig. 3 include: - "Oh right, I see there is **a** sort here..." and "I guess there is **a** sort of some kind here which involves a swap." Phrases like 'that's probably', or "That is a…" also suggest that the hypothesis is code-prompted.

The inference-based episode can be considered a cycle of scanning the code, inferring the abstracted hypothesis, and validating that hypothesis.

3.3. Identifying Bottom-up Comprehension.

Bottom-up comprehension can be interpreted as a line-by-line description of code leading to eventual confirmation of plan.

Again, using the code in Fig. 3, and 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 bottom-up comprehension process can generally be characterised by the following episode: initial study, leading to aggregation, and finally the programmer derives a conclusion.

4. Results, Analysis & Discussion.

4.1. Results.

As mentioned earlier, two independent coders were used to code the verbal data protocols. Following the coding process, a joint meeting was held between the two independent coders. Here the differences between the two coders were discussed, and any difficulties encountered during the coding process were examined. The results were subsequently reconciled and are shown as follows: -

Reconciled Results
Insurance Company

Familiar Program

Subject	Expectations	Inferences	Bottom-up
DB	2	7	2
IB	3	3	1
PH	1	2	1
SB	6	8	3
	12	**20**	**7**

Unfamiliar Program

Subject	Expectations	Inferences	Bottom-up
DB	0	3	1
IB	0	3	0
PH	0	4	0
SB	2	3	2
	2	**13**	**3**

Reconciled Results
Healthcare Company

Familiar Program

Subject	Expectations	Inferences	Bottom-up
BK	1	3	3
BH	3	0	1
DT	3	5	6
GB	1	3	3
	8	**11**	**13**

Unfamiliar Program

Subject	Expectations	Inferences	Bottom-up
BK	0	7	0
BH	0	8	0
DT	0	5	1
GB	0	3	0
	0	**23**	**1**

These results suggested a high level of agreement between the coders. However, to ensure that the same phrases were being coded, 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 3.

Table 3 – Coders Consistency results per phrase (all categories)

	Coder 1	Coder 2
Consistent	108	108
Inconsistent	5	5

This table identifies the amount of phrases the coders produced, which they agreed, and disagreed on. It turns out that they were in agreement over 95% of the time. However this figure consists of bottom-up phrases as well. Just considering the pre-generated expectations and inferences alone, the figures are presented in Table 4.

Table 4 – Coders consistency results per phrase (in terms of expectation-based & inference-based)

	Coder 1	Coder 2
Consistent	85	85
Inconsistent	4	4

Here a slight drop is shown, but still around 93%.

4.2 Results Analysis & Discussion.

Our first aim was to differentiate and establish the co-existence of the two processes. Given that 93% of the phrases were consistently categorised with respect to the coders' manual. This strongly suggests that both methods of plan triggering were in existence in the sessions.

To statistically analyse our findings with respect to familiarity, we used the Mann-Whitney-Wilcoxon test. This test is a non-parametric test and is used to compare small sample sizes such as ours. Results from this test are shown in the following three tables.

Expectations

Mann-Whitney U	3.500
Wilcoxon W	39.500
Z	-3.147
Asymp. Sig. (2-tailed)	**.002**
Exact Sig. [2*(1-tailed Sig.)]	.001

The first tested the difference between familiar and unfamiliar programmers' usage of expectations during the comprehension sessions. There is a significant difference here with (P = .002) for the two-tailed test and (P = .001) for the one-tailed test. As expected, programmers familiar with the domain mostly relied on expectations. This is because they, being familiar with the domain, have a greater knowledge of that domain itself, and are thus able to pre-generate expectations about typical plans from that domain.

Inferences

Mann-Whitney U	25.500
Wilcoxon W	61.500
Z	-.714
Asymp. Sig. (2-tailed)	**.475**
Exact Sig. [2*(1-tailed Sig.)]	.505

The second tested the difference between familiar and unfamiliar programmers' usage of inferences during comprehension. The difference here was insignificant (P=.475). The figures suggest the inferences were used by both groups fairly equally. Although this was expected, it was anticipated that programmers who were familiar would be more likely to infer program plans, whereas this was not the case. One suggestion for the surprising result is that unfamiliar programmers were forced to make more inferences as they were not the reservoir of pre-generate expectations from their unfamiliar domain.

Bottom-up

Mann-Whitney U	6.500
Wilcoxon W	42.500
Z	-2.771
Asymp. Sig. (2-tailed)	**.006**
Exact Sig. [2*(1-tailed Sig.)]	.005

The final test tested the difference between familiar and unfamiliar programmers' usage of bottom-up processing. There is a significant difference here (P=0.006). One possibility is that this may be due to familiar programmers, going through the program a second time, having more time to fully get a feel for its meaning and therefore may have worked through it methodically on a line-by-line basis.

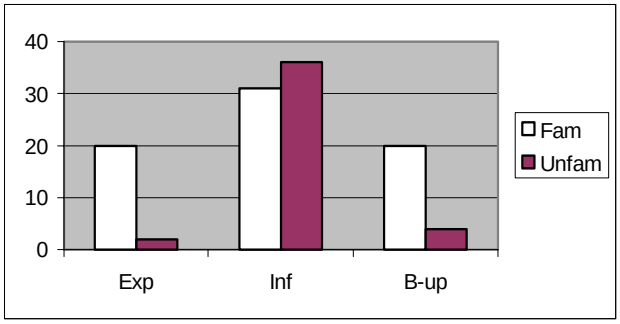

Fig. 4 – Subjects' overall usage of variants with respect to domain familiarity

In summary, from the results obtained during this experiment, it was clear that expert programmers, during program comprehension, employ all three types of comprehension processes and that familiarity led to increased use of pre-generated expectations. Fig. 4 shows the total amount of expectations, inferences, and bottom-up processing, used by programmers from both the familiar and unfamiliar domains.

5. Conclusion.

This paper described an experiment that investigated inference-based processing, expectation-based processing and systematic-based processing by programmers during program comprehension. We categorised talk-aloud responses in a manner similar to that carried out by Shaft, thus distinguishing between bottom-up and top-down methodologies. However, this work extended Shaft's categorisation, identifying two distinct types of top-down comprehension, namely, inference based where the programmer derives meaning from clichéd implementations in the code, and expectation-based comprehension where the programmer has pre-generated expectations of the code's meaning.

From our experiments, we discovered that expert programmers employ all three strategies during program comprehension. However, if the domain is familiar, these programmers will mostly rely on expectations and inferences during understanding.

In contrast, programmers unfamiliar with the domain did not rely on expectations during comprehension. This was because they had no expertise in this domain, and therefore were not in a position to pre-generate expectations about the code's meaning. These programmers used inferences quite a lot, possibly due to their language expertise.

6. Acknowledgements.

The authors wish to thank Dr. Chris Douce for his valuable suggestions and constructive criticism on earlier versions of the paper. They would also like to thank Alan Sheahan and Dr. Oliver Hyde, for their statistical expertise. Special thanks must go to Professor Teresa Shaft for giving them her permission to use and alter her coders' manual.

7. References.

[1] De Lucia A., Fasolino A. R., Munro M., "*Understanding Function Behaviours through Program Slicing*". Proceedings of the 4th Workshop on Program Comprehension, pp 9-19, 1996

[2] Rajlich V., "*Program Reading and Comprehension*", Proceedings of Summer School on Engineering of Existing Software, pp 161-178, 1994

[3] Shaft T. M., "*Helping Programmers Understand Computer Programs: The Use of Metacognition*", Database Advances, Vol. 26, No. 4, pp 25-46, 1995

[4] Von Mayrhauser A., Vans A., "*Program Understanding Behaviour during Debugging of Large Scale Software*", Proceedings of Empirical Studies of Programmers: 7th Workshop, pp 157-179, 1997

[5] Von Mayrhauser A., Vans A. M., "*Program Understanding: Models and Experiments*", Advances in Computers, Vol. 40, No. 4, pp 25-46, 1995

[6] Brooks, R., "*Towards a Theory of the Comprehension of Computer Programs*", International Journal of Man-Machine Studies, Vol. 18, pp 543-554, 1983

[7] Pennington, N., "*Comprehension Strategies in Programming*", Empirical Studies of Programmers: 2nd Workshop, p 100, 1987

[8] Letovsky, S., "*Cognitive Processes in Program Comprehension*", Empirical Studies of Programmers: 1st Workshop, p 58, 1986

[9] Soloway, E., "*Empirical Studies of Programming Knowledge*", IEEE Transactions on Software Engineering, IEEE Computer Society, Vol. SE-10, No. 5, Sep. 1984

[10] Lind, R., Vairavan, K., "*An Experimental Study of Software Metrics and their Relationship to Software Development Effort*", IEEE Transactions on Software Engineering, SE-15, 5 (1989), pp 649 - 653.

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

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

[13] Buckley, J., Cahill, A., "*Measuring Comprehension Behaviour Through System Monitoring*", Proceedings of the Workshop on Empirical Studies of Software Maintenance, 1997

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

[15] Tilley, S., Paul, S, Smith, D., "*Towards a Framework for Program Understanding*", 4th International Workshop on Program Comprehension, 1996

[16] Storey M. A-D, Fracchia, F, Muller, H., "*Cognitive Design Elements to Support the Construction of a Mental Model During Software Exploration*", The Journal of Systems and Software, Vol. 44, 1999