Empirically Studying Software Practitioners – Bridging the Gap between Theory and Practice

Michael P. O'Brien, Jim Buckley, Chris Exton
Department of Computer Science & Information Systems
University of Limerick
michaelp.obrien@ul.ie, jim.buckley@ul.ie, chris.exton@ul.ie

Abstract

It is the view of many computer scientists that the standard of empirical software engineering research leaves scope for improvement. However, there is also an increasing awareness in the software engineering community that empirical studies are a vital aspect in the process of improving methods and tools, for software development and maintenance.

This paper presents a review of the empirical work carried out to date in the area of program comprehension and illustrates that most of the evidence from these studies derives from lab-based experiments, thus implying a degree of artificial control. The paper argues that, in order to address the methodological shortfalls of the experimental paradigm, more qualitative methods need to be applied to accompany and support these quantitative studies, thus broadening the sources of data and increasing the 'body of evidence'.

1. Introduction

In general terms, a study is referred to as 'empirical' if it involves observing or measuring something. For many years empirical studies have been considered an important tool for understanding software development practices, and also for giving valuable information on how to improve practice. All empirical studies require suitable methodologies and techniques, and this is no different when studying programmers.

This paper examines some fundamental psychological perspectives and how they have prompted a bias towards formal experimental methodologies in the area of software comprehension studies (when this might not be optimal with respect to the maturity of the field). Many would argue that observational methods would seem more appropriate for a new discipline such as this. Gilmore [11], for example, suggests that findings from initial observational studies could be used as the basis for larger, more quantitative, studies due to their ability to capture more complex aspects of programmer behaviour. However, given the current situation, observational in-situ studies could still be used to evaluate claims from existing formal studies of program comprehension, thus broadening the evidence base. This is particularly important given the lesser external validity associated with formal experiments.

Essentially, validity refers to the meaning of experimental results and more specifically to the agreement between participants' measured scores and the attribute under study [15]. Perry et al. [4] identify three types of validity important in empirical software engineering studies:

- External validity, which concerns making generalisations about results obtained from a study. In other words, how well do the conclusions of a study apply to other people, in other places, at other times. Admittedly, it is not often possible to obtain information from every individual in a particular population and researchers have little option but to draw 'samples' from the larger population. They can then generalise the results back to the entire population of interest using appropriate statistical techniques.

- Internal validity, which is concerned with the possibility that there may be other possible causes or explanations, which resulted in observed behaviour other than the hypothesized 'cause'. In other words, an experimenter should have evidence that the independent variables in

Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05)
1063-6773/05 $20.00 © 2005 **IEEE**

the study caused what was observed, rather than any other 'non-related' factors.

- Construct validity, which essentially, assesses the degree to which the dependent variables measure the theoretical constructs under study. In other words, construct validity is an assessment of how the experimenter translated their ideas or theories into actual measures.

Much of the empirical work carried out to date involves tightly controlled experiments in an artificial environment. These studies alone may not provide sufficient evidence to gain an insight into the general behaviour of industrial programmers, mainly due to their low level of external validity.

This paper begins by presenting some fundamental psychology perspectives that provide the basis for various kinds of empirical design. Section 3 describes empirical studies in the context of software comprehension and argues that much of the work to date is based on a cognitive psychology perspective, with its strong association with experimental methods. Section 4 looks at the two main research methods used when carrying out empirical studies, namely, the qualitative approach, and the quantitative approach. It examines each method in terms of the psychology perspectives identified in section and argues that there is a growing potential for a broader, more comprehensive, methodological approach to the study of software comprehension.

2. Psychological Perspectives

Psychology perspectives offer a wide forum for empirical validation. This section of the paper discusses the relationships between three fundamental psychology perspectives. For each perspective, the kinds of evidence they seek, is described, along with the description of these 'evidence-types'. Comparisons of these perspectives can shed light on similar concerns in the area of software comprehension studies and the authors of this article argue that knowledge of these perspectives can be a very useful resource when designing empirical studies of programmers. Intuitively, there seems to be no exclusively correct perspective and each is useful in its own right for seeking supporting evidence for theories of software comprehension. This evidence can co-exist and merge to form a more rich and complete body of evidence.

2.1. The Behaviorist Perspective

The first of these perspectives is 'behaviorism', where understanding a given situation is achieved through observation of an individual's behaviour [1]. Behavioural theories reject the notion that individuals universally pass through a series of stages. Instead, people are assumed to be affected by the environmental stimuli to which they happen to be exposed.

The central pillar of the behavioural perspective and its associated method is its strict adherence to scientific principle; only controlled observable and quantifiable experimentation could be considered as an acceptable source of knowledge. This need to be scientifically thorough is a major influence on the type of methods used by behaviorists. Watson (cited by [1]) for example, one of the early and influential advocates of the behavioural perspective, felt that it was impossible to gain any scientifically useful insights into the mind by using introspection. Only an outsider (objective) viewpoint could be considered when data was collected.

A further major influence on the methods used by behaviorists was the acceptance that much of our behaviour is shared with many other animals, in particular, vertebrates, and specifically, non-human primates. For example, psychologists studying types of instrumental conditioning use an experimental environment such as a maze (193 , cited by [1]) with randomly selected rats to test their various hypotheses.

This perspective centers itself solely around the 'behaviour' of an individual rather than how their brain actually carries out, processing. Although not directly related to program comprehension, behaviorism serves to highlight the drawbacks of an extreme scientific approach to understand the functionality of the human brain; hence evolved the cognitive perspective.

2.2. The Cognitive Perspective

The cognitive perspective somewhat rejects the notion that one can fully understand an individual by primarily observing their behaviour. Instead, this perspective focuses on the internal processes that allow people to know, understand, and think about the world. The essential challenge here is to gain insight into what is going on inside a person's head. In other words, the central aim is to understand the complex

functions such as reasoning, memory, problem solving, or the decision-making processes that occur inside the individual's head. A major influence on the methods used by cognitive psychologists is the view that the human brain may be viewed as a computer [5]. They would argue that, in the same way as we cannot hope to gain an understanding of a software algorithm solely by observing its inputs and outputs, we cannot hope to comprehend the complex workings of the human mind by merely observing the stimulus-response data gathered by behaviorists. It is perhaps due to this analogy with computing [ ] that this perspective is nearly exclusively the basis for most research into software comprehension.

The research methods of behavioural and cognitive perspectives have much in common, including the fact that they both focus on the individual and primarily rely on formal, controlled experiments. However, without the restriction of the scientific fundamentalist doctrine associated with behaviorism, the methods employed by the cognitive perspective have diversified from the sole use of data that is gathered from outwardly observable behavior, to the 'insider' data that is acquired through verbal reports or introspection [ ] and data that is generated from various brain imaging techniques [30].

It is the belief of cognitive psychologists that the comparative assumptions made by behaviorists in relation to animals break down when one considers higher order functions of the brain, which essentially is the area, which mostly interests those in the cognitive field. From our review of the literature, is seems true that most empirical studies in the area of software comprehension (see section 3) adopt this perspective intuitively in their design, and many cognitive models have been proposed, which aim to characterize the process of software cognition.

2.3. The Socio-Cultural Perspective

Essentially the socio-cultural perspective underscores the need to look at *real activities* in *real situations* [1]. Socio-culturists consider external (ecological) validity to be a central requirement in any study, and therefore their methods are mainly observational based and are qualitative in nature. It should be noted that quantitative data may be produced later by performing a detailed analysis on the data using appropriate coding categories, but the methods used in socio-cultural studies could not be described as mainly quantitative in nature. The socio-cultural perspective characteristically considers the individual in a social and environmental context, with comprehension viewed as a process of enculturation that involves both physical and psychological tools, such as computers and language. This differs markedly from the behaviorist and cognitive perspectives, as they are primarily concerned with the individual. Socio-culturists would argue that the actual cognitive activity involved in solving many problems is directly affected by the tools the candidate uses to find the solution; making the environmental context a central component in their choice of methods. Therefore, in empirical studies of programmers, researchers should pay careful attention to gaining a full understanding of, and document, the software tools, which programmers use to aid the process of understanding code.

Socio-culturists would also argue that any methods that study an individual in isolation from their community would also fall short in terms of providing valuable insights into how we understand, because much of our understanding is through a process of discussion and debate. Thus socio-cultural psychologists would consider that research methods focusing solely on the individual in a purely experimental environment are deficient and incapable of gaining a true insight into how understanding occurs. In fact, most empirical work carried out to date does not take into consideration external validity in terms of the programmers' environment, the source code used in these studies, or the tasks required of participants.

In summary, socio-culturists would assert that understanding is not just about gaining knowledge; it is a process of enculturation that provides us with a suitable lexicon and shared understanding that enables us to engage with others about discipline or subject area.

3. Empirical Work in Software Comprehension Studies

While there exists three dominant psychological perspectives, much of the empirical research done in the software comprehension domain, falls into the cognitive perspective. The socio-cultural perspective, with its emphasis on environmental and cultural factors seems like an ideal complementary perspective with which to study software comprehension. This is because such an approach places emphasis on external validity, a factor, which may be lessened in more formal controlled experiments.

This section presents some of the empirical work located in our review of empirical studies of software comprehension (a fuller review can be obtained from the primary author on request). The review illustrates the quantitative experiment-methodology adopted in this area and highlights the various control issues that lessen external validity, suggesting that the studies' relevance to industrial programming practice is difficult to defend. In many of these studies, no requirements or design documentation were provided. An overview of empirical work in this area is presented in Table 1, followed by a more detailed review of three such studies.

Table 1 – **External Validity Concerns in Program Comprehension Studies**

Reference	**Controlled Variables (Summary)**
[3]	Environment - Laboratory Task - Summary & Quiz Code - Program segments (presented in hardcopy format)
[35]	Environment – Laboratory Task - Fill in the blanks Code - program segments (in hardcopy format)
[35]	Environment – Laboratory Task - Recall Code - Program segments (in hardcopy format)
[17]	Environment – Laboratory Task - Enhancement task Code - 50 lines
[33]	Environment – Laboratory Task - Fill in the blanks Code - Program segments (hardcopy)
[19]	Environment – Laboratory Task - Cloze test, quiz, blind summary Code - 50 LOC (in hardcopy format)
[5]	Environment – Laboratory Task – Recall verbatim Code - Small BASIC programs
[7]	Environment – Laboratory Task - Evaluate expressions Code – Expressions (presented in hardcopy format)
[4]	Environment – Laboratory Task - Comprehension quiz Code - 16 program segments (presented in hardcopy format)
[0]	Environment – Laboratory Task - Debugging Code - Pascal program (73 LOC)
[10]	Environment - Laboratory Task - Summary Code – small Pascal programs (in hardcopy format)
[6]	Environment - Laboratory Task - Group related lines of code Code - 1 programs (1 -4 LOC) all presented in hardcopy format
[4]	Environment - Laboratory Task - Recall Code - Pascal program (3 LOC) presented in hardcopy format
[7]	Environment - Laboratory Task - Segment and label sections & Group programs Code - 1 small programs (35-57 LOC presented in hardcopy format)
[ ]	Environment - Laboratory Task - 'Scrambled' lines of code to be comprehended Code - Pascal program (136 LOC presented one line at a time on a video display terminal)
[3]	Environment – Laboratory Task - Recall & quiz Code – COBOL program (67 LOC in hardcopy format)
[6]	Environment – Laboratory Task - Post-comprehension quiz Code - 4 small C++ programs (in hardcopy format)
[1]	Environment – Laboratory Task - Recall Code - 5 line BASIC program (presented in hardcopy format)
[36]	Environment – Laboratory Task - Quiz 30 true/false questions Code - Two 1 line Pascal programs (in hardcopy format)
[ ]	Environment - In-situ Task - Retrospective summary Code - industrial programs (presented in hard copy format)
[39]	Environment - In-situ Task - Observational study Code - 4 industrial programs (60-100K LOC)
[41]	Environment - In-situ Task - Observational study Code - Large scale (industrial)

3.1. Example Study 1: Soloway & Ehrlich

Soloway & Ehrlich's empirical studies [35] assess the role of plans and programming discourse rules in the comprehension processes of expert programmers. Essentially, programming plans can be defined as "program fragments that represent stereotypic action sequences in programming to achieve a specific goal" [6]. Rules of programming discourse then represent clichéd code conventions that are used in the development of programming plans. Soloway & Ehrlich hypothesize that experts built and maintain programs using both knowledge of these plans and the rules of programming discourse.

Soloway's first experiment used a 'fill-in-the-blank' technique. 139 students were used as participants in this experiment. They were presented in hardcopy format, with a series of small program segments in which a selected line of code was omitted. Participants had no indication as to what the program actually did, but were asked to insert a line of code that best completed it, based implicitly on the (plan/unplan like nature of the) code surrounding that line.

The psychological quality of the design of this empirical study (i.e. cloze testing) is well supported and interesting information emerged, namely, that expert programmers use plan knowledge when comprehending source code. What many would question here is the reality and industrial relevance of the findings, as rarely is it the case that software practitioners are presented with small code segments to insert omitted lines of code.

In the second study, the only difference was that this time 41 expert programmers, with a mean number of 7.5 years experience, were used as participants, and were given a recall task rather than a fill-in-the-blank type exercise. Participants were given a suite of -10 line program segments, one after another. Participants were presented with each program 3 times, each time for 0 seconds.

However, the controlled nature of the experiment again leads to so external validity concerns. Rarely, if ever, are industrial programmers required to *recall* verbatim (after a series of 0 second glances), lines of source code that have been taken away. Also, the code 'segments' used in both studies are unrealistic in terms of what industrial software engineers work with on a daily basis. Industrial programmers deal with large systems, which can sometimes contain many tens of thousands of lines of code.

3.2. Example Study 2: Pennington

Pennington's work [3] centres on mental representations of source code, and her theory of program comprehension incorporates a program model and a domain (situation) model. She identified five categories of information, obtainable from source code:

- *Operations* – information about a specific action in the code
- *Control flow* – information about sequences of operations during execution
- *Data flow* - information about data structures, data transformations and dependencies
- *State* - information about the state of all pieces of a program when a certain point in execution is reached
- *Function* - information about what the program actually does in terms of its domain

Pennington hypothesized that the first mental representation built, is the control flow abstraction of the program. She refers to this as the "program model", which is built bottom-up using control structures to identify/classify blocks of code in the program text. Following partial construction of this model, a more domain-oriented model is created – the situation model. This representation is also built in a bottom-up fashion where the current program model is extended to encompass the data flow/functional based abstractions. When the overall goal is reached, the situation model and understanding, is complete.

Pennington demonstrated this theory by carrying out two experiments. The first experiment involved 0 professional programmers (40 COBOL programmers & 40 FORTRAN programmers) who studied short program texts (15 lines approximately). Pennington used short program texts so as to achieve a high degree of experimental control. These participants were required to answer a series of questions, and undergo a 'priming test'. This priming test was used to examine mental distances between program statements in the programmers' minds. Each of the programmers was presented with a specific sequence of statements, some of which were contained in the code segment, and others, which were not. Their task was to differentiate between the two. The comprehension quiz involved a series of questions to assess what information types (as identified earlier) could be attained in a limited amount of time. Equal numbers of questions were generated to

assess the programmers' knowledge of each of the five different information categories.

In the second study, participants studied a 00-line program, and again, were asked a series of questions. This time, response-latency and accuracy was measured. 40 programmers were chosen from the existing group (0 COBOL programmers and 0 FORTRAN programmers). These participants were programmers (of each language) who had scored in the top and bottom quartiles in the comprehension task of the previous experiment. They were required to answer comprehension questions on each of the different abstraction types, and to carry out a modification task to an existing program. The test began with 45 minutes study time. Participants were then asked to write a summary of the program, which was immediately followed by 0 comprehension questions. After that followed 30 minutes modification time. When this session concluded, participants were asked another 0 questions and to write a summary of the program.

Regarding the validity of Pennington's work, here again, participants in these studies were presented with relatively small code segments, unlike in industry where programmers work with much larger systems. Also, industrial programmers are rarely, if ever, presented with a scenario following a maintenance task, where they have to recall whether or not certain lines of code, presented to them on paper, were contained in the program text, as in Pennington's first experiment. It could also be argued that it would be unusual for professional programmers to be quizzed on their knowledge of the five information categories defined earlier without the presence of the source code or executing system.

3.3. Example Study 3: Shneiderman & Mayer

Shneiderman & Mayer's empirical work [33] builds upon fundamental cognitive psychology beliefs, held, on the way human memory is organised. Essentially, they hypothesize that the programmer takes code statements into short-term memory. Syntactic knowledge (knowledge of the programming language at hand) is then brought from long-term memory to develop a low-level understanding of these code constructs. Semantic knowledge (general programming knowledge) is then brought from long-term memory to match the syntactic constructs and identify their function. This knowledge is built-up by chunking to form higher-level semantic units.

To test 'part' of their theory, Shneiderman & Mayer conducted an experiment, which used the "fill-in-the-blank" technique. 4 programmers (4 novices and 4 advanced programmers) were used as subjects for the first study. Subjects were given small segments of Fortran code which contained either logical "IF" or arithmetic "IF" statements. This study essentially concentrated on subjects understanding of these arithmetic and logical "IF" statements. In several code segments, a space was left for the programmer to fill in. On filling in this line correctly, it was assumed that the subject had gained a full understanding of the condition. Following this fill-in-the-blank exercise, subjects were asked a series of questions relating to the code segments.

Again, if one examines the external validity of these findings, many socio-culturists would question the industrial relevance of this 'fill in the blank technique' and the use of small code segments. Indeed, they would also argue that it would be most unusual for industrial programmers to be faced with a scenario where they are asked to fill in the missing line of a code segment and indeed work in an artificial environment away from their peers.

3.4. Validity Concerns

Some computer scientists dismiss these empirical studies as "ineffective". They argue that they are a shallow and artificial view of the nature of programming skill [31], [ ]. This pessimism is possibly due to the fact that most of these studies tend to ask participants to carry out an unrealistic task (in terms of it's relevance or similarity to industrial work), for example, filling in missing lines of code [purposely omitted] in a code segment [35], [33] at this level of granularity. Obviously when using a CASE tool, auto generation can occur but usually at a framework level. Also, experiments are often carried out in a laboratory setting rather than an industrial setting, which again, questions the ecological validity of their findings.

A further issue that leads to dismissal of empirical work is the realism of the source code used (mainly presented to the participant in hardcopy format - see Table 1), along with the enforced time limit constraints in understanding it. As seen in Table 1, many experiments use 'code segments' or very small programs to enhance experiment control. In fact several studies use unfamiliar code to 'ensure' a level starting point for each programmer. However, on a daily basis, industrial programmers are faced with partially familiar systems from partially familiar

domains. These systems often exceeding hundreds of thousands of lines of code in size and are viewed through specialized IDEs.

3.5. Observational Studies of Software Practitioners

Some studies, however, tried to adopt a more qualitative, observational approach. Von Mayrhauser & Vans [37] present a meta-model of software comprehension, which integrates Pennington's 'program' and 'situation' models [3], with Soloway & Ehrlich's model of software comprehension [35]. Essentially, this model evolved from empirical studies, carried out by von Mayrhauser & Vans, which concluded that programmers use a combination of strategies when understanding code.

To form and validate this meta-model, Von Mayrhauser et al [3], [39], [40], [41], carried out observational studies on experienced professional programmers while they enhanced / maintained software, and analyzed their behaviour. The objectives of this study were to:

- gain insight into the kinds of actions programmers perform when enhancing code;
- identify whether or not programmers used a specific comprehension process;
- identify the types of hypotheses programmers use and how they are resolved.

At the heart of these studies was an attempt to find out if programmers followed the meta-model. Programmers were observed in-situ as they carried out enhancements to software, which exceeded 40,000 lines of code and their talk-aloud data was captured as they worked.

The studies themselves had a high degree of external validity in the sense that many factors were taken into consideration during their design. Firstly, professional industrial programmers with many years experience programming large systems were employed as participants. Secondly, large systems were used, which contained over 40,000 lines of code - typical of what maintenance engineers work with. Thirdly, the systems used were those that the programmers worked on normally. Fourth, the enhancement tasks were real and thus typical of those faced on a daily basis by professional programmers. Finally, the studies were ecologically valid as it took place in an industrial setting, which was familiar to the participants (although it is unclear if participants used their community when undertaking their task, or if they would normally do so).

Our research review identified this series of studies as unique in adopting an observational approach (with high external validity) in this area.

4. Qualitative & Quantitative Approaches

Empirical work can be categorised based on its ability to produce numeric output. An empirical study that produces numeric output is termed quantitative and one, which produces (possible rich) textual output, is referred to as qualitative [9]. One major difference between the two approaches is that qualitative research is inductive and quantitative research is deductive [1]. Typically, however, quantitative studies have been associated predominantly with the cognitive perspective and qualitative research, with the socio-cultural perspective. Thus, this qualitative / quantitative breakdown can provide additional evidence for the need to shift to more observational studies.

Buckley [3] interestingly points out that in the proceedings of the 'Empirical Studies of Programmers' (ESP) workshops to date, quantitative studies out-number qualitative studies and studies with qualitative elements combined (see Table).

Table 2 - **Quantitative & Qualitative Studies in ESP [3]**

ESP	No. of quantitative studies reported on	No. of qualitative studies reported on	No. of studies with elements of both
19 6	11	3	3
19 7	6	4	5
1991	9	4	3
1993	4		5
1996	10	0	7
1997		1	4
Total	**48**	**14**	**2**

This again supports the argument for a greater use of qualitative, socio-culturally research methods to

increase our understanding of the software processes [9] (even though the qualitative / quantitative breakdown is not an exact match for socio-cultural / cognitive). For example, several studies have derived quantitative information from qualitative data [13], [14]. These research methods greatly help us investigate vaguely defined concepts like understanding and learning, all of which are concepts that are central to the improvement and development of tools that facilitate the comprehension process. Snyder [34] insists, that researchers can simultaneously employ qualitative and quantitative methods if studies are carefully designed and carried out conscientiously.

5. Conclusion

According to Good & Brna [14], one of the main criticisms levelled at program comprehension studies is that "they do not capture the richness of the activities which occur in a naturalistic setting". Lakhotia [16] also states that much of the empirical work in this area is typically carried out in controlled environments using program 'segments' and claims that the conclusions drawn from these studies are debatable. Others would argue that the limited level of industrial realism of their design has put a question mark on the external validity and generality of much of the empirical work carried out to date in the area of program comprehension.

In contrast, some researchers would argue that a qualitative socio-cultural approach does not contain the same rigor and replication aspects that are common to most quantitative methods. However, one fails to see how many of the insights and observations that are a product of qualitative research may be brought to light by formal experiment-based approaches without a large loss of related contextual depth and associated richness of data.

The paper argues that the most convincing claims are those that are built on multiple, independent, but converging strands of research from both the qualitative and quantitative domains, and in essence, these two methodological approaches need each other to derive mutual support. However, because typically qualitative data involves words and quantitative data involves numbers, there are some researchers who feel that one is inherently better than the other. In conclusion this paper suggests that there is a mandatory role for both qualitative and quantitative approaches if we hope to gain a more complete and realistic understanding of industrial programming behaviour.

6. Acknowledgements

This work was undertaken as part of the Science Foundation Ireland Investigator Programme, B4-STEP (**B**uilding a **B**i-Directional **B**ridge **B**etween **S**oftware **Th**E**o**ry and **P**ractice: http://www.b4step.ul.ie). The authors wish to thank Dr. Pär J. Ågerfalk (University of Limerick) for valuable critique & suggestions on earlier drafts of this paper.

. References

[1] Barfield, W. "Expert-Novice Differences for Software: Implications for Problem-Solving & Knowledge Acquisition", *Behaviour and Information Technology*, Vol. 5, No. 1, pp 15- 9, 19 6

[ ] Brooks, R. "Towards a Theory of the Comprehension of Computer Programs", *International Journal of Man-Machine Studies*, Vol. 1 , pp 543-554, 19 3

[3] Buckley, J. "System Monitoring: A Tool for Capturing Software Engineers' Information-Seeking Behaviour", *PhD Thesis*, University of Limerick, 00

[4] Cunniff, N., Taylor, R. P. "Graphics and Learning: A Study of Learner Characteristics and Comprehension of Programming Languages", *Human-Computer Interaction – INTERACT '87*, Elsevier Science Publishers, 19 7

[5] Davies, S. P. "The Nature and Development of Programming Plans", *International Journal of Man-Machine Studies*, Vol. 3 , pp 461-4 1, 1990

[6] Davies, S. P. "Expertise and the Comprehension of Object-Oriented Programs", *Proceedings of the 12ih PPIG Workshop*, Italy, 000

[7] Eisenberg M., Resnick, M., Turbak, F. "Understanding Procedures as Objects", *Proceedings of the Empirical Studies of Programmers: 2nd Workshop*, Ablex Publishing, 19 7

[ ] Eysenck, M., Keane, M. *'Cognitive Psychology: A Student's Handbook*", Psychology Press, UK, 1995

[9] Finkelstein, A., Kramer, J. "Software Engineering: A Roadmap", *Proceedings of the Conference on the Future of Software Engineering*, Limerick, 000

[10] Gellenbeck, E., Cook, C. "An Investigation of Procedure and Variable Names as Beacons During Program Comprehension", *Technical Report 91-60-2*, Oregon State University, 1991

[11] Gilmore, D. "Methodological Issues in the Study of Programming", In Hoc, Green, Samurcay & Gilmore (Editors), "*Psychology of Programming*", London, Academic Press, 1990

[1] Glaser, B. G., Strauss, A. L. "*The Discovery of Grounded Theory: Strategies for Qualitative Research*", Chicago, 1967

[13] Good, J. "Programming Paradigms, Information Types and Graphical Representations: Empirical Investigations of Novice Program Comprehension", *PhD Thesis*, University of Edinburgh, 1999

[14] Good, J., Brna, P. "Towards Authentic Measures of Program Comprehension", *Proceedings of the 15th Annual PPIG Workshop*, Keele, UK, 003

[15] Kaplan, R. M., Saccuzzo, D. P., "*Psychological Testing: Principles, Applications & Issues*, 3rd Edition, 1993

[16] Lakhotia, A. "Understanding Someone Else's Code: Analysis of Experiences", *Journal of Systems Software*, Vol. 3, pp 69- 75, 1993

[17] Letovsky, S. "Cognitive Processes in Program Comprehension", *Proceedings of Empirical Studies of Programmers: 1st Workshop,* 19 6

[1] Littleton, K., Toates, F., & Braisby, N. "Chapter 3: Three Approaches to Learning", *Mapping Psychology 1*, Milton Keynes, The Open University, 165- 16, 00

[19] Littman D.C., Pinto, J., Letovsky S. and Soloway E. "Mental Models and Software Maintenance". *Empirical Studies of Programmers: 1st Workshop*, pp 0-9 , 19 6

[0] Nanja, M., Cook, R. C. "An Analysis of the On-line Debugging Process", in *Empirical Studies of Programmers, 2nd Workshop*, Ablex Publishing, 19 7

[1] Nardi, B. A. "Studying Context: A Comparison of Activity Theory, Situated Action Models and Distributed Cognition", In Nardi, B. A. (Ed.) *Context and Consciousness: Activity Theory and Human-computer Interaction*. Cambridge, MA: MIT Press, 1996

[ ] O'Brien, M., Buckley, J. "Inference-based & Expectation-based Processing in Program Comprehension", *Proceedings of the 9th International Workshop on Program Comprehension*, Toronto, Canada, 001

[3] Pennington, N. "Stimulus Structures and Mental Representations in Expert Comprehension of Computer Programs", *Cognitive Psychology*, Vol. 19, pp 95-341, 19 7

[4] Perry, D., Porter, A., Votta, L. "A Primer on Empirical Studies", *Tutorial Presented at the International Conference on Software Maintenance,* 1997

[5] Pinker, S. "*How the Mind Works*", New York: Norton, 1997

[6] Rist, R. "Plans in Programming: Definition, Demonstration, and Development", *Proceedings of the 1st Workshop on the Empirical Studies of Programmers*, Ablex Publishing, pp -47, 19 6

[7] Robertson, S. P., Yu, C. C. "Common Cognitive Representations of Program Code Across Tasks & Languages", *International Journal of Man-Machine Studies*, Vol. 33, pp 343-360, 1990

[ ] Robertson, S. P., Davis, E. F., Okabe, K., Fitz-Randolf, D. "Program Comprehension Beyond the Line", *Human-Computer Interaction, INTERACT '90*, Elsevier Science Publishers, 1990

[9] Seamen, C. "Qualitative Methods in Empirical Studies of Software Engineering", *IEEE Transactions of Software Engineering*, Vol. 5, No. 1, pp 557-57 , 1999

[30] Šešok, S., Jensterle, J. "Psychological Methods and the Study of Cognitive Function", *Proceedings of IS2001*, Slovenia, 001

[31] Sheil, B. A. "The Psychological Study of Programming", *ACM Computing Surveys*, Vol. 13, No. 1, pp 6 9-707, 19 1

[3] Shneiderman, B. "Measuring Computer Program Quality and Comprehension", *International Journal of Man-Machine Studies*, Vol. 9, pp 465-47 , 1977

[33] Shneiderman, B., Mayer, R. “Syntactic / Semantic Interactions in Programmer Behaviour”, *International Journal of Computer and Information Sciences*, Vol. , No. 3, pp 19-33 , 1979

[34] Snyder, I. Multiple Perspectives in Literacy Research: Integrating the Quantitative and Qualitative, *Language and Education, 9* (1), 45-59, 1995

[35] Soloway, E., Ehrlich, K. “Empirical Studies of Programming Knowledge”, *IEEE Transactions on Software Engineering*, Vol. SE-10, No. 5, pp 595-609, 19 4

[36] Teasley, B. “The Effects of Naming Style & Expertise on Program Comprehension”, *International Journal of Human-Computer Studies*, Vol. 40, pp 757-770, 1994

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

[3] Von Mayrhauser, A., Vans A. M. “Industrial experience with an integrated code comprehension model”, *IEEE Software Engineering Journal*, Vol. 10, No. 5., 1995

[39] Von Mayrhauser A., Vans A., "Identification of Dynamic Comprehension Processes During Large Scale Maintenance", *IEEE Transactions on Software Engineering*, Vol. , No. 6, 1996

[40] Von Mayrhauser, A., Vans, A.M. “On the role of hypotheses during opportunistic understanding while porting large scale code”, *4th Intl. Workshop on Program Comprehension*, 1996

[41] Von Mayrhauser A., Vans A., Howe A.E. "Program Understanding Behaviour during Enhancement of Large Scale Software", *Software Maintenance: Research and Practice*, Vol. 9, pp 99-3 7, 1997

[4] Wiedenbeck, S. “Beacons in Computer Program Comprehension”, *Intl. Journal of Man-Machine Studies*, 5, pp 697-709, 19 6