Characterizing Software Comprehension for Programmers in Practice[1]

Jim Buckley
B4-Step Group
University of Limerick
Ireland
Jim.Buckley@ul.ie

Abstract

One of the main problems associated with empirical studies of programmers is their ecological validity. Participants are often asked to work on unfamiliar systems for a short period of time, using unfamiliar browsing environments. To compound this, they may be required to perform tasks that are unrepresentative of their daily work. This does not suggest sloppy empirical design. Ecological validity is extremely difficult to obtain, especially in the context of formal quantitative experiments where strict controls limit the variability that is normally associated with work contexts. However, this limitation does call into question the relevance of the findings for software practitioners.

The research currently being carried out at the University of Limerick aims to address this difficult ecological-validity issue, specifically in the context of software comprehension studies. We intend to observe programmers performing their job in their working environment and to see the tasks they perform. Subsequently, we intend to identify the knowledge they bring to these tasks and how they use it. Our intention is to perform one-subject studies and to form partnerships with other researchers allowing them to replicate the studies with a high degree of confidence in different contexts. This will allow the community to assess the generality of the findings.

Keywords: Software Comprehension, Empirical Studies, Ecological Validity

1. Introduction

Several researchers [Von Mayrhauser and Vans '95], [Good '99], [O'Brien '01] have described software comprehension as consisting of four core components: the knowledge base that programmers bring to comprehension, the external representations available for them to study, the mental model that they form of the system, and a processing element (See Figure 1). The processing element uses the external representations available and the knowledge base of the programmer (including their existing mental model of the system) to expand, refine and correct their current mental model of the system.

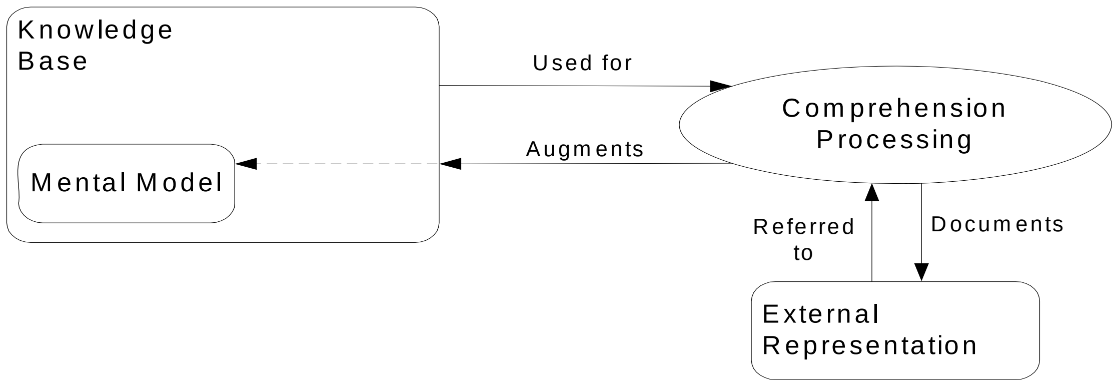

Fig 1. Elements of Software Comprehension.

[1] This work is being funded by the SFI under the auspices of the B4-Step Research Program

Many researchers have probed the nature of the programmers' knowledge base. For example [Brooks '83] proposed that programmers' use domain knowledge, general knowledge and algorithmic knowledge when developing systems. He suggested that the job of the maintainer when comprehending a system was to re-build the bridges that the developer created between these bodies of knowledge. He proposed that this process was essentially top-down where high-level domain hypotheses were broken down into lower level hypotheses until they could be matched to the code. This classification of knowledge types and knowledge use has subsequently been used by [Von Mayrhauser and Vans '96, '97], to (empirically) characterize the knowledge that programmers use during code comprehension.

[Gellenbeck and Cook '91], [Soloway and Ehrlich '84], [Wiedenbeck '97], [O'Brien and Buckley '01] and [Boehm-Davis et al. '96] have all studied the plan structure of software systems, a plan being a clichéd (possibly de-localized) section of code that implements a specific functional goal [Rist '86]. [Gellenbeck and Cook '91], [Soloway and Ehrlich '84] [O'Brien and Buckley '01] and [Wiedenbeck '97] worked on identifying how clichéd signals (or beacons) present in code allow the programmer to hypothesize on the presence of a plan in the code and how they subsequently attempted to validate these hypotheses. [Boehm-Davis et al. '96] and [Rist '86] tried to identify the structure of plans that programmers impose on source code.

[Letovsky '86] implicitly, and later [Detienne '02] explicitly, suggested that programmers' mental models were structured as knowledge schemas. A schema can be considered a framework, or data structure, which organizes knowledge in memory [Detienne '02]. This framework may be filled with data instances as code is studied. For example, the programmer may realize that part of the code calculates a person's tax-free allowance. This will suggest a schema to the programmer, which is only partially filled with data, based on their current knowledge. The programmer may then decide to fill the remaining, empty slots in the data structure by studying the code. In this instance, for example, they may seek out the variable that holds the person's salary or the variable that holds their number of dependents. This structure seems closely related to the concept of plans as described above.

Other research has concentrated on the processes that programmers perform as they begin to understand software systems. [Pennington '87], for example, proposed that programmers use abstraction to chunk the system into a lesser number of more-manageable [Miller '56] mental blocks. Initially, these blocks reflect the control-block structure (while loops, for loops, if constructs) of the program. 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 model maps the (possibly de-localized) events, data-states and structures of the source code to the events, states and structures of the real world [Detienne, '02].

2. Tying Research to Practice

This research could be a valuable resource for industry. For example, by identifying the importance of domain knowledge and schemas, there is an implicit assumption that domain experience would be highly useful when maintaining code. By highlighting the role that programmers' knowledge of clichéd plans and beacons play in software comprehension, strong guidelines with respect to implementation standards could be derived [Gellenbeck and Cook '91].

2.1. Current Issues

Software practitioners seem reluctant to adopt these findings, and we suggest that part of the reason for this is the low ecological validity [Thomas and Kellogg '89] of the empirical studies that underpin the findings. Programmers in industry often spend years maintaining different parts of large software systems. In contrast many of the empirical studies mentioned here have introduced programmers to unfamiliar software systems (or segments of software) for a very short period of time[2]. This is typically done to achieve control over programmers' familiarity with the system. If familiarity is held constant, then the relationships between other variables can be assessed with higher internal validity [Perry et al. '97]. However, this restriction means that such studies cannot hope to assess the growing familiarity that programmers gain with their individual software systems over time. Instead they can only inform research on programmers' initial time with a software system.

Another ecological validity issue that might lower the relevance of empirical findings for programmers is that of task. Many empirical studies require participants to perform unrealistic tasks. Programmers, for example, have been asked to recall the code verbatim [Wiedenbeck '97], provide summaries of the code [O Brien and Buckley '01], answer specific questions when the code is taken away [Pennington '87], [Ramalingham and Wiedenbeck '97] and cluster code segments [Boehm-Davis et al '96]. While these tasks can certainly inform psychologists, and the academic community, on the underlying mental models and comprehension processes of participants, we imagine that these tasks are all highly unrepresentative of the tasks demanded of programmers in their normal working lives. Thus, findings from these studies will be treated with suspicion by practitioners.

For an illustrative counter-example, consider [Littman et al '86]'s study of debugging which found that several participants actively avoided building up a systematic understanding of the system when presented with a seemingly localized bug. In fact, these programmers seemed to avoid large tracts of the system altogether, a finding replicated by [Buckley '02] and [Nanja and Cook '87].

2.2. Research Agenda

These issues are not easy to address. If research were to be carried out in naturalistic working environments, researchers would have limited control over the programmer's familiarity with the system, their familiarity with the domain and their task context. Thus variables could not be as strictly controlled and interference in cause-effect relationships between dependent and independent variables would be difficult to discount. However, if empirical research is to have relevance to practitioners, then the ecological validity issue must be addressed.

[2] To our knowledge only two exceptions exist. [O'Brien and Buckley '01] allowed programmers study a medium sized [Von Mayrhauser and Vans '95] program from their own company, but the protocol used did force them to study it using a hard copy. [Von Mayrhauser and Vans '96, '97]'s empirical work is of a higher ecological validity, allowing programmers to work on their own systems, doing their own tasks, with their own normal exploration tools in their own environment.

Initially, industrial partners will be solicited for observational studies. These studies will report on the tasks performed by programmers and the manner in which they are performed. From these observations, working hypotheses will be developed as to the way in which programmers comprehend code and documentation during their work. These hypotheses will be the basis for one-subject studies [Harrison '97] that will be performed with programmers in their natural working environment, as they work. The studies will report on whether individual programmers act in a manner that supports the hypotheses and will allow us to assess the findings from previous, more-formal quantitative studies in a real-world context.

While we do not rule out quantitative studies [Seamen '99], we also intend to gather quantitative data that can be added to by independent researchers who replicate our work in different organizations. In order to increase the potential for replication, and thus build a body of evidence for our hypotheses, we must ensure that our protocol and analysis methods are open, clear and transparent to other researchers.

We assume a vast heterogeneity in the population of programmers and their work environment. Such heterogeneity can be useful in assessing the generality of findings when similarities appear across different programmers in different organizations. However, we anticipate that often, similarities will not be apparent across these boundaries, and in order to determine causality, strong characterization mechanisms for the workplace and for the programmers must be established. This is one of the first tasks envisaged under this program of work.

References

Boehm-Davis D.A., Fox J.E. and Phillips B.H. (1996)."Techniques for Exploring Program Comprehension" in Empirical Studies of Programmers: Sixth Workshop, Ed: Gray and Boehm-Davis. Ablex Publishing Corporation. pp 3-38.

Brooks R. (1983). "Towards a Theory of Comprehension of Computer Programs" International Journal of Man-Machine Studies. Vol: 18, pp 543-554.

Buckley J. (2002). "System Monitoring: A Tool for Capturing Software Engineers' Information-Seeking Behavior" PhD Thesis. University of Limerick

Detienne F. (2002). "Software Design – Cognitive Aspects" Springer-Verlag. ISBN: 1852332530.

Gellenbeck, E.M. and Cook C.R. (1991). "An investigation of Procedure and Variable names as Beacons during Program Comprehension." Technical Report 91-60-2, Oregon State University.

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

Harrison W. (1997). "Editorial". Empirical Software Engineering. Vol 2. no. 1. pp 7-10.

Letovsky S. (1986). "Cognitive Processes in Program Comprehension" in Empirical Studies of Programmers. Eds: Soloway and Iyengar. Ablex Publishing Corporation. ISBN 0-89391-388-X pp 28-45.

Littman D.C., Pinto J., Letovsky S. and Soloway E. (1986). "Mental Models and Software Maintenance." in Empirical Studies of Programmers. Eds: Soloway and Iyengar. Ablex Publishing Corporation. ISBN 0-89391-388-X pp 80-98.

Miller G.A. (1956). "The Magic Number 7, Plus or Minus Two: Some Limits on our Capacity for Processing Information." Psychological Review. Vol: 63 pp 81-93.

Nanja M. and Cook C. R. (1987). "An Analysis of On-Line Debugging Process." Empirical Studies of Programmers: Second Workshop. Eds: Olsen, Sheppard and Soloway. Ablex Publishing Corporation. pp 100-113.

O Brien M.P. (2001) “Software Comprehension of Familiar Systems”. MSc Thesis, Limerick Institute of Technology.

O Brien M.P. and Buckley J. (2001). "Inference-Based Comprehension and Expectation Based Processing in Program Comprehension." in Proceedings of the 9th International Workshop on Program Comprehension. pp 71-78.

Pennington N. (1987). "Comprehension Strategies in Programming." in Empirical Studies of Programmers: Second Workshop. Eds. Olson, Sheppard and Soloway. Ablex Publishing Corporation. pp 100-114

Perry D., Porter A. and Votta L. (1997). "A Primer on Empirical Studies". Tutorial Presented at the International Conference on Software Maintenance.

Ramalingam V. and Wiedenbeck S. (1997). "An Empirical Study of Novice Program Comprehension in the Imperative and Object Oriented Styles" in Empirical Studies of Programmers: Seventh Workshop. Eds. Wiedenbeck and Scholtz. ACM Press. pp124-140

Rist R. (1986). "Plans in Programming: Definition, Demonstration and Development" in Empirical Studies of Programmers. Eds: Soloway and Iyengar. Ablex Publishing Corporation. pp 28-45.

Seamen C. B. (1999). "Qualitative Methods in Empirical Studies of Software Engineering". IEEE Transactions of Software Engineering. Vol: 25, no 4. pp 557-572.

Singer J. and Lethbridge T. (1996). "Methods for Studying Maintenance Activities" The Workshop for Empirical Studies of Software Maintenance.

Soloway E. and Ehrlich K. (1984). "Empirical Studies of Programming Knowledge". IEEE Transactions on Software Engineering. Vol: 10, no: 5 pp 565-609.

Thomas, J.C. and Kellogg, W.A.. (1989). "Minimizing Ecological Gaps in Interface Design". IEEE Software. Vol: 6 no: 1. pp: 78 -86

Von Mayrhauser A. and Vans A.M. (1995). "Program Understanding: Models and Experiments" Advances In Computers. Vol: 40 pp 1-38.

Von Mayrhauser A. and Vans A. (1996). "Identification of Dynamic Comprehension Processes During Large Scale Maintenance". IEEE Transactions on Software Engineering. Vol: 22, No. 6, pp 424-437.

Von Mayrhauser A. and Vans A.M. (1997). "Hypothesis Driven Understanding Processes During Corrective Maintenance of Large Scale Software". Proceedings of the International Conference on Software Maintenance. Eds: Harrold M.J. and Vis-aggio G. pp 12-20.

Wiedenbeck S. (1997). "Processes in Computer Program Comprehension" in Empirical Studies of Programmers, Ed: Soloway and Iyengar. Ablex Publishing Corporation. pp 187-198.