The Stores Model of Code Cognition

Christopher Douce

Institute of Educational Technology
Open University, Walton Hall, Milton Keynes, UK
c.douce@open.ac.uk

Keywords: POP-II.B. program comprehension, POP-V.A. short-term memory

Abstract

Program comprehension is perhaps one of the oldest topics within the psychology of programming. It addresses a central issue: how programmers work with and manipulate source code to construct effective software systems. Models can play an important role in understanding the challenges developers and engineers contend with. This paper presents a model of program comprehension, or code cognition, which has been derived from literature found within the disciplines of computing and psychology. Drawing on direct experimentation, this paper argues that a model of code cognition should take account of the visual, spatial and linguistic abilities of developers. The strengths and weaknesses of this model are discussed and further research directions presented.

1. Introduction

Program comprehension is a topic that has been studied using different approaches. One approach is to turn to engineering and ask the question 'what can we build to support engineers in the work that they do?' Researchers have constructed elegant software visualisation tools (Young, 1996), debuggers, integrated development environments and program slicers (Weiser, 1981). After the introduction of new tools (and techniques) the efficiency of software engineering practice can be measured empirically to determine whether an advantage is gained in terms of time and cost. One difficulty lies with the relentless pace of development in the field of software development. By the time such studies are established, new innovative tools and techniques are likely to emerge.

Another approach is to study the programmer directly to learn what they do and the strategies they apply. Understanding what occurs during comprehension has the potential to guide the development of tools and techniques. This paper explores program comprehension from this second perspective, a psychological perspective, where the focus of enquiry is the programmer.

This paper presents a simple model of code cognition called the 'stores model'. This model is derived from two sources: observations obtained during a series of historical experimental tasks[1] and research that has studied the operation and structure of working memory. A driving principle behind creating the stores model is to attempt to connect experimental observations to research carried out within field of cognitive psychology.

The next section of this paper briefly presents some important program comprehension research. This is followed by a description of the studies that were drawn upon to form the basis of further exploration. The fourth section briefly presents some relevant background that was uncovered during the course of the research. This is followed by a presentation of the stores model. Its strengths and weaknesses are considered in section six which is followed by a number of conclusions and further research directions.

[1] Historical in the sense that the tasks presented here were carried out within an earlier project. The results from these tasks have been used to inform the construction of the stores model.

2. Program Comprehension

One of the earliest models of program comprehension was proposed by Shneiderman and Mayer (1979) who proposed that there is a separation between syntactic and semantic knowledge. Syntactic knowledge, namely how to punctuate and structure programming code is considered to be somewhat fragile and can be easily forgotten, whereas semantic knowledge represents deep understandings of key programming concepts and ideas.

One of the earliest studies of comprehension was carried by Green (1977) that explored the efficiency of different types of 'programming notation'. This study and others were conducted to unpick earlier assertions that some programming constructs were easier to understand than others. Writing at a similar time, Brooks drew from work on production rules and then later proposed the notion that 'programming beacons' might play a part during comprehension (Brooks, 1983). Beacons, it was argued, are stereotypical fragments of code that could be recognised by a programmer and subsequently provide hints about what a program might be doing. Using beacons, programs could be comprehended in a 'bottom up' manner and were subject to empirical study by Weidenbeck (1986). The notion of stereotypes and the awareness of the importance of pre-existing knowledge inspired the application of schema theory as a way to understand program comprehension. This can be seen in the work of Soloway and Ehrlich (1984), Detienne (1990) and Rist (1989).

Other researchers approached the problem in different ways. Aware that programming is a skilled task and requires expertise Pennington asked the question, 'what representations do programmers use when they work with software?' and constructed a number of comprehension tests to attempt to determine whether there were any differences between how programmers understood code (Pennington, 1987). This has parallels with Green's work on programming notations (Green, 1990).

Accepting the importance of programmer knowledge, some researchers became aware of the importance of strategy: how programmers make use of their experience knowledge, the problem and the code to understand elements of a software system and solve problems. Davis presented a literature review of the field (Davis, 1993), and other researchers later considered how and why comprehension occurred using top-down, bottom-up, systematic and opportunistic strategies (O'Brien, Buckley & Shaft, 2004).

Attempting to 'bring together' the different strands of program comprehension research, von Mayrhauser and Vans (1995) devised a new 'integrated meta-model'. This model attempted to combine different conceptions of program comprehension and present them within a single model. The von Mayrhauser and Vans model draws heavily from software engineering and computer science literature and less from work originating from psychology. Although successful in presenting a picture of 'comprehension state of the art' it appears to be somewhat distant from the intention of uncovering which aspects of cognition are applied during the solving programming problems.

To understand what occurs during program comprehension the most ecologically valid approach to adopt is to observe how programmers work 'in the wild'. Programmers conduct problem-oriented research using websites, construct designs using paper prototypes, attend meetings and contribute to discussion forums. Comprehension can occur during coding, code reading or exploration and periods of software maintenance. In essence, comprehension can be considered to occur as a part of wider activities.

In a more recent study, Ko et. al. (2006) have explored program comprehension using a combination of debugging and program enhancement tasks carried out within a popular integrated development environment. As a result of this work, it was discovered that a significant amount of time was spent manipulating and navigating the source code. In doing so, Ko et. al. propose a 'new model as a process of searching, relating and collecting information of perceived relevance in which the development environment plays a central role' (Ko. et. al., 2006, p. 972).

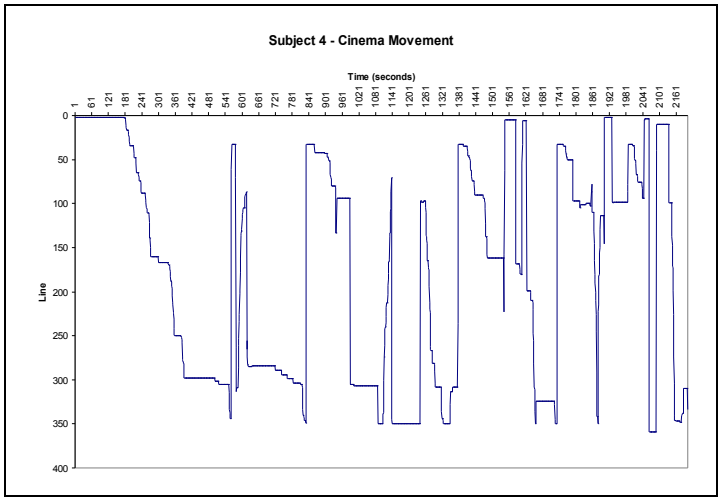
It is recognised that significant extraneous variables can potentially affect how certain aspects of program understanding can be studied. As a result, researchers have often attempted to manage the environment in which activities take place. The next section describes an observational study of

programmers who have been recruited from an academic environment. The results of this study has inspired the generation of a program comprehension model, the stores model, which draws from the work of others.

3. Studies

Eight subjects were directed to carry out minor enhancements to six different object-oriented programs. The programs were self contained, required no substantial third party libraries or components. Each program was operated through a command line interface. All participants were familiar with the programming language but were unfamiliar with the structure of the programs. The program editing and browsing activities were captured through an instrumented text editor. Each participant was asked to talk aloud during the maintenance activities. Further descriptions regarding the tasks can be found in an earlier paper (Douce & Layzell, 1999).

The data was collected with the intention of gaining an understanding of the strategies that programmers could use to maintain simple object-oriented program. A starting point for analysis was the versions of the edited programs and the cursor movement data. When combined with line number information this data provides a snap shot of which portions of a program was especially useful. The following graph illustrates the movement of the cursor throughout an entire listing of one of the problems during a maintenance task:



Graph 1 : Subject 4, Cinema modification

The cursor position provides an approximate indication as to which parts of the program were considered to be useful or informative. The notion of focal points has been of interest to other researchers, particularly Romero et. al. (2004) who used a code browsing utility to clearly capture which elements of a program were of interest during certain tasks.

The movement graphs for all of the subjects were surprising. Movements appeared to be unexpectedly rapid. This graph shows a progression from ‘scanning’ reading to fast, rapid access of code fragments. The subject steadily moves from the top of the program to the bottom in small, controlled movements. When the parts relevant to the maintenance task have been identified and partially comprehended, movements appear direct and precise.

It is at this point that it is worth asking, ‘how does a programmer do this?’, or ‘what helps a programmer to do this?’ The notion of positional memory is familiar. Imagine if your house has a power cut. You can probably find your way around your house or apartment without too much difficulty. You may be able to recall the position of the cupboards in your kitchen and the location of your torch.

Spatial ability can be described to be the ability to manipulate spatial relations, and it is suggested that such abilities play an important role during program comprehension and maintenance. The impressive speed that the cursor moves over the surface of a program suggests that programmers possess a capacity for remembering and quickly returning to landmarks or *spatial beacons*. Spatial beacons can be described as fragments of code that may have special importance in relation to comprehension or a related task. A beacon has the potential to be an intermediate location or a destination of its own right. The importance of these locations have been acknowledged by tool developers, where development environments such as Visual Studio enable developers to bookmark or place anchors at particular points within a source text, allowing immediate navigation between different locations.

Knowing the location of code within a program is only one issue. Code comprises of identifiers, symbols and structures that can be shared across different files. A programmer must be able to read linguistic symbols to be able to relate these to different spatially located sections of code.

In one instance it was apparent that spatial knowledge of the program was not used. To locate items one subject preferred to conduct a number of lexical search operations rather than to remember precisely where in the source text code fragments were positioned. Spatial working memory was essentially bypassed since the locations of the code fragments were derived from the search operations.

Program comprehension is understood to be a multifaceted activity. Understanding what comprehension is requires an integration of different abilities. One of the underlying themes within the comprehension literature is, of course, the role of memory. It is this theme that we explore within the next section of the paper.

4. Working Memory Model

Alan Baddeley developed what is called the working memory model due to dissatisfaction of the simplistic notion that the human memory system could be broadly divided into two components; a long-term store and a short-term store (Baddely, 1997). This dissatisfaction was caused by the inability of the model to explain emerging experimental evidence from experimental psychology and cognitive neuropsychology. The Baddeley working memory model comprises of three main components: a central executive that is said to resemble attention and two slave components, the visuo-spatial sketchpad and a phonological loop.

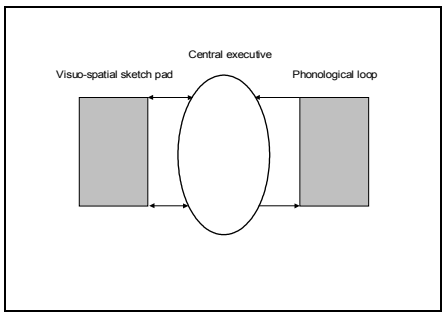


Figure 1 : Adapted from Baddeley, 1997

The phonological loop is closely related to the articulatory system, a cognitive system that is used to process and generate language. Out of all the components of memory, the phonological loop is considered the most thoroughly understood. It is said to comprise of two smaller components; a temporary store that is used to store ‘speech-based information’, and an articulatory control process which is said to refresh the temporary store.

The phonological loop has the ability to accept written material and convert it into a phonological code, hence allowing it to enter short-term memory. The visuo-spatial sketch-pad allows visual imagery to be ‘constructed’ and spatial information to be held. Visual imagery was subject to a

revival within experimental psychology during the 1960's. An illuminating paper by Paivio discusses the relationship between the construction of mental images and the memorability of words, and presents a stimulating history of mental imagery within psychology (Paivio, 1969).

The sketchpad has some similarities to the phonological loop. As mentioned, information within the loop can come from two sources; through the audio channel or via written words. Information within the sketchpad can come from the perceptual or visual system, or from the system that constructs mental imagery.

Evidence is gradually mounting that visuo-spatial sketchpad system may be comprised of two smaller systems which have been shown to have a neurological basis; a system that is concerned with the detection of *what*, and a second system that is concerned with the *where*. The 'what' and 'where' distinction can be found in a very early description of a brain injury (Holmes, 1919). Holmes described a subject who had difficulties eating: his subject could correctly identify what a spoon was along with its purpose, but had difficulties locating it in space. It was evident that when an object had been grasped, the subject could return to that location without difficulty, using information obtained from position of the body through preprioception. Later, Levine, Warach and Farah (1985) also illustrated the 'what' and 'where' distinctions. They described a case where a subject was unable to draw or describe objects held in memory, but showed good ability to locate items in space.

The central executive component, the central component can be described as more an attentional system rather than a memory store. Researchers have proposed several theories of attention, but these take a step towards the challenging arena of 'consciousness' and what it may be. Like the other components of the model, neurological evidence has played a role in its creation. A subject with a unique form of deficit can be found within the work of Baddeley and Wilson (1985).

To summarise, the working memory provides an elegant way to consider different aspects of human cognition. The model has been influential since its work is grounded from research originating from two different areas of psychology. It is worth asking the question, 'how does this relate to programming and program comprehension?' Programming is obviously an activity that necessitates the use of memory. This working model can explain some of the actions necessary to carry out programming: reading code and navigating through 'code space'.

Like the work carried out by von Mayrhauser and Vans, the following section aims to connect the model that is described within this section to some of the models of comprehension that were described earlier.

5. Stores model of code cognition

The three components of the working memory model can be clearly seen in the middle of the below diagram. Additional components that have been drawn from other researchers are presented surrounding the model.

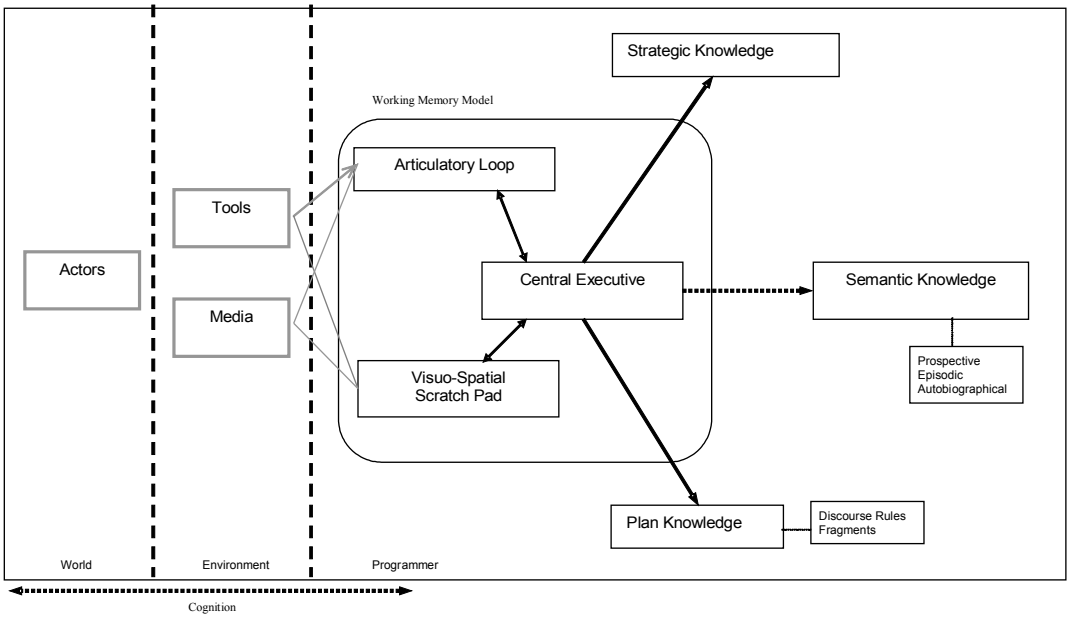


Figure 2 : The Stores model of code cognition

The central executive plays a key role. This component dominates code problem solving. There is an important relationship between it and the strategic knowledge component. Strategic knowledge presents guidance to the central executive about how to integrate different sources of information. This knowledge will provide pointers to both high level comprehension strategy, such as top-down, bottom up or opportunistic, and recommend low level tactical guidance such as how and where to encode information, specifically, which cognitive system to use. The conception of a programming strategy can be associated with the notion of a cognitive strategy. Certain problems can be solved either using spatial or verbal approaches (Roberts, Gilmore & Wood, 1997). This understanding could be extended to more sophisticated (and ill defined) domains such as program comprehension.

Semantic knowledge bears some relationship to the work by Shneiderman and Mayer. Semantic knowledge, it can be argued, can be split into further categories: autobiographical memory, episodic memory and prospective memory. Whether such distinctions are useful and real is subject to on-going debate. Semantic knowledge is considered to be knowledge about a problem domain, internalised information about requirements and general software development knowledge.

Plan knowledge, drawn from the earlier work on plans and beacons is considered to be similar to semantic knowledge. Effective code is constructed by following discourse rules which relates to what is acceptable within a particular coding environment. Plan knowledge can contain stereotypical fragments that can be applied within the context of a software development programme.

Connections between the slave components of working memory and the components *media* and *tools* are proposed. 'Tools' refers all the systems and software development environments that a developer can use. This includes search utilities and engines, debuggers, source management software and integrated development environments. In essence, anything that can be used to manipulate and find out about code can be placed within the 'tools' box. 'Media' on the other hand, refers to materials that are outside the 'digital world' of a computer. Media can refer to designs on paper, annotated printouts, interface sketches, post it notes or an eXtreme programming 'wall' (Beck, 1999). Tools

such as integrated development environments can present users with compilation messages which have to be read. Errors may be directly linked to a corresponding location within a source file.

The use of external media has been explored by Davis (1991) who studied the differences between externalisation of information by novice and expert programmers. Researchers have identified cases where external notations can act as a temporary 'buffer' for working memory. Pen and paper drawings and diagrams can be used as a way to both retrieve and store both linguistic and spatial information. The choice of when to move artefacts from internal memory to external memory (and how these are represented) will be made by drawing on existing strategic knowledge and an awareness of meta-cognitive state.

The final component of the model, the *actors* component, refers to individuals and organisations that make up the wider arena in which software is constructed. Software development is an activity that requires collaboration between individuals. Programming can be a group activity as much as it is an individual activity.

6. Strengths and Weaknesses

One of the main strengths of the model is that it could be said to be grounded in research that has attempted to understand how people work with, memorise and use different forms of information. The model, therefore, has the potential to be used to further understand low-level programming practice. The use of different types of human memory as a focal point is, however, not a new idea, and is an important part of Walentstein's HASTI cognitive re-engineering framework (Walenstein, 2002).

The stores model is incomplete. Little attention has been given to episodic or prospective memory both of which could play an important role within large-scale program comprehension tasks. Episodic memory is memory of past actions. Remembering what fragments of source code were being viewed yesterday and where these program fragments were located in any number of files is a faculty which is undoubtedly useful. Prospective memory, on the other hand, refers to memory about having to do things. Prospective memory is undoubtedly the most obvious form of memory failing; forgetting to visit the dry-cleaners on the way home or forgetting to buy an essential ingredient for a recipe from a market, are two examples. Programmers may postpone comprehension or modification of particular aspects of a software system until a later date, perhaps due to changing organisational pressures. A fascinating review of research conducted into this area can be found in a paper written by Morris (1992).

The stores model has drawn on the work of other comprehension models as sources of inspiration. Any problems that exist within the other models have also been brought into the stores model. Like the working memory model, the central executive remains the most mysterious component, and schema theory is not without its critics. One other criticism of the stores model is that it is not directly apparent where to map some of the earlier ideas regarding program comprehension. The notion of syntax and semantics being separate does not have an immediate parallel within the stores model, but instead may be dispersed over different components.

The strength of the links between each of the components of the model need to be tested. Does the link between the central executive and the other components have any substance? One of the strengths of the model is that it invites further experimentation and study. One of its weaknesses is that, whilst it might be useful to guide the development of future studies, it inevitably asks more questions that it solves.

Finally, the model only represents a tiny part of programming cognition. Source code is not only comprehended at an individual level, but within teams and organisations. The stores model is situated within the cognitive tradition of psychology of programming research and its use may be limited to researchers who are examining the equally important social dimensions of software development.

7. Conclusions

The stores model takes inspiration from older models of program comprehension as posited by computer science and human factors researchers and integrates them with other developments from cognitive psychology. There are parallels between the work of von Mayrhauser and Vans with the idea of combining different models together.

The model has been partly developed from the desire and need to explain behaviour observed during experiments to explore the maintenance of object-oriented programs. It appeared to be the case that subjects moved through code at phenomenal speeds, at a rate that was entirely unexpected. It was therefore suggested that some form of memory was being utilised to hold on to 'code location'. Such a store was believed to hold spatial information about the code that was being manipulated. Baddeley's working memory model provided one approach to attempt to explain what was being observed.

The validity of the model needs to be explored. It also suggests a number of interesting research directions. Possibilities include whether spatial memory plays an important role for the long-term comprehension of software systems, and whether the sub-types of memory that have been proposed are useful during code comprehension and manipulation.

The presentation of the model raises the interesting question of whether the stores model could be realised or implemented in some way, perhaps drawing upon ACT-R or GOMS for inspiration (Anderson, 1993; Card, Moran & Newell, 1983; Altman, 2001). The act of building a computational representation of an abstract model requires the developers to ask difficult questions about how elements of a system should be structured and operate.

The subject of program comprehension it is considered to be an important area and focus of study, but in recent years the interest in comprehension as a discrete subject has waned. This said, the IEEE Workshop on Program Comprehension continues to play an important role and there remains much to be gained from continuing to explore the connections between software engineering and psychology. Whilst it is tempting to create engineering solutions to try to ease the challenges of comprehension, focus should still fall on programmers and software developers. It is important that we continue to try to understand what they do and how they solve the problems that they face.

5. References

Anderson, J. R. (1993). Rules of the Mind. Erlbaum: Hillsdale, NJ

Altman, E. M. (2001) Near-term memory in programming: a simulation based analysis. International Journal of Human-Computer Studies, 54, 189-210.

Baddeley, A. (1997) Human memory : theory and practice - Revised edition. Psychology Press : Hove

Baddeley, A. & Wilson, B. (1986) Amnesia, autobiographical memory, and confabulation, in Autobiographical memory, D. Rubin, Editor. 1986, Cambridge University Press: New York.

Beck, K. (1999) Extreme programming explained: embrace change. Addison-Wesley.

Brooks, R. (1983) Towards a theory of the comprehension of computer programs. International Journal of Man-Machine Studies, 18, 543-554.

Card, S. K., Moran, T. P. & Newell, A. (1983) The psychology of human-computer interaction. Lawrence Erlbaum Associates: Hillsdale, NJ.

Davis, S.P (1991) Externalising information during coding activities: effects of expertise, environment and task. in Empirical Studies of Programmers : Fifth workshop. New Brunswick, NJ.: Ablex.

Davis, S. P. (1993) Models and theories of programming strategy. International Journal of Man-Machine Studies, 39, 237-267.

Detienne, F. (1990) Expert programming knowledge : A schema-based approach, in Psychology of Programming, J.-M. Hoc, T.R.G. Green, and R. Samurcay, Editors, Academic Press: London.

Douce, C. R. & Layzell, P, J. (1999). Evolution and errors: An empirical example. IEEE International Conference on Software Maintenance, Oxford, England.

Green, T. R. G. (1977) Conditional program statements and their comprehensibility to professional programmers. Journal of Occupational Psychology, 50, 93-109.

Green, T. R. G. (1989). Cognitive dimensions of notations. In Sutcliffe, A. and Macaulay, L., editors, People and Computers V, Cambridge University Press, 443–460

Holmes, G. (1919) Disturbances of visual space perception. British Medical Journal. 2, 230-233.

Levine, D.N., Warach, J. & Farah, M. (1985) Two visual systems in mental imagery: dissociation of 'what' and 'where' in imagery disorders due to bilateral posterior cerebral lesions. Neurology, 35, 1010-1018.

Ko, A. J., Myers, B. A., Coblenz, M. J. & Aung, H. H. (2006) An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. IEEE Transactions on Software Engineering, 32(12), 971-987.

Morris, P.E. (1992) Prospective memory: remembering to do things, in Aspects of Memory : Vol 1., M. Grunenberg and P. Morris, Editors, Routledge: London, 196-222.

O'Brien M., Buckley J. & Shaft T. (2004). Expectation-based, inference-based and bottom-up software comprehension. Journal of Software Maintenance and Evolution: Research and Practice, 16(6), 427-447.

Paivio, A.(1969) Mental imagery in associative learning and memory. Psychological review, 76, 241-263.

Pennington, N. (1987) Stimulus Structures and Mental Representations in Expert Comprehension of Computer Programs. Cognitive Psychology, 19(3), 295-341.

Pirolli, P. & Card, S. K. (1999) Information foraging. Psychological Review, 106, 643-675.

Rist, R.S., (1989) Schema creation in programming. Cognitive Science, 13, 389-414.

Roberts, M. J., Gilmore, D. J., & Wood, D. J. (1997). Individual differences and strategy selection in reasoning. British Journal of Psychology, 88, 473-492.

Romero, P., du Boulay, B., Cox, R., Lutz, R., & Bryant, S. (2004). Dynamic rich-data capture and analysis of debugging processes. In Dunican, E. & Green, T. R. G. (Eds.), Proceedings of the 16th annual workshop of the Psychology of Programming Interest Group, 140-150.

Shneiderman, B. & Mayer, R. (1979) Syntactic/semantic interactions in programmer behavior: a model and experimental results, International Journal of Computer and Information Sciences, 8(3), 219-238.

Soloway, E. & Ehrlich, K. (1984) Empirical studies of programming knowledge. IEEE Transactions of Software Engineering. SE-10(5), 595-609.

von Mayrhauser, A. and Vans, A. M. (1995). Program comprehension during software maintenance and evolution. Computer, 28(8), 44–55.

Walenstein, A. (2002) HASTI: A Lightweight Framework for Cognitive Reengineering Analysis. Proceedings of 14th Annual Workshop of the Psychology of Programming Interest Group, 219-234.

Weidenbeck, S. (1986) Beacons in computer program comprehension. Interational Journal of Man-Machine Studies, 25, 697-709.

Weiser, M. (1981) Program slicing. Fifth International Conference on Software Engineering. IEEE.

Young, P. (1996) Software visualisation. Technical Report, University of Durham : Centre for Software Maintenance.