TOWARD A THEORY OF PROGRAMMING SCHEMES

Jane Terry Nutter
Tulane University

Introduction

People have been writing programs and program documentation for about forty years now. For just as long, people have had to read these program texts. Because programming languages were initially designed for machines, not people, understanding programs presents special problems which persist despite the move toward high level languages and structured programming. With program documentation, the problem is perhaps worse: while programming languages become progressively more "English like", documentation has been moving in the opposite direction. Pseudo code and graphical representations are replacing natural language explanations, suggesting that understanding programs may not be particularly like understanding familiar natural language texts at all.

The fundamentally dynamic nature of programs supports this suggestion. Ordinary natural language texts certainly include dynamic aspects (verbs, for instance!), but nouns provide a stable element which helps anchor meaning. Programs conspicuously lack nouns. A difference of this magnitude must affect how we understand them. Yet understanding program texts must also resemble understanding natural language texts, since learning to deal with programs draws on previously developed skills, some of them reading skills. I believe that understanding in general is guided by abstract concepts, and that program understanding shares this trait. But in the case of program understanding, the concepts in question are abstract patterns for dynamic activities, which I call programming schemes.

Schemes and Representations

A programming scheme is an abstract structure which captures the essential features of a dynamic process to solve some problem. These schemes contain the conceptual information of what a particular chunk of code does. But because they are abstract, they are never directly present in any particular programming text. A particular instantiation in some concrete form is a scheme representation. The underlying patterns (schemes) provide templates by which people understand programs containing embodiments of them (scheme representations).

Researchers in natural language understanding have long believed that abstract cognitive patterns guide text understanding. Schank and Abelson (1977) hold that scripts govern story understanding. Schank (1982) proposes that memory organization packets (MOPS) and thematic organization points (TOPS) underlie memory and cognition. Similarly Chase and Simon (1973), Shneiderman (1976) and Adelson (1981) argue that experts use meaningful abstract formations in their domain of expertise to recall things they were given to memorize.

Researchers have also begun to investigate the role of schemes or something like them in programmer behavior. Soloway has undertaken extensive work in this direction (see e.g. Soloway and Woolf, 1981; Soloway, Ehrlich, Bonar and

Greenspan, 1982; Ehrlich and Soloway, 1982; and Soloway, Ehrlich, and Gold, 1983). Further evidence that abstract constructs govern programming behavior can be found in work by Weiser (1982), Curtis and Sheppard's group (see e.g. Curtis, Sheppard, Milliman, Borst, and Love, 1979 and Sheppard, Milliman, and Love, 1979), and Magel (1982). Yet even Soloway's extensive work leaves schemes themselves largely unanalyzed. We cannot use programming schemes to explain how people understand program texts unless we understand programming schemes. To date, no clear account of what programming schemes are, how they are related to one another, how they are related to their representations, or how they are identified has been given. Hence we have neither a theory to unify these results nor a model for predicting human interactions with particular schemes or their representations.

Why a Theory?

An analysis of programming schemes offers several benefits. First, it should produce a clearer characterization of schemes. To say that a programming scheme is an abstract process template is suggestive, but little more. What properties of programming schemes distinguish them from other abstract concepts and objects?

Second, it should clarify the kinds of possible relationships among schemes. For instance, schemes can contain other schemes: the "merge and sort" scheme contains the "file merge" scheme and the "file sort" scheme. But the "merge and sort" scheme is not an instance of the "file merge" scheme, nor is the "file merge" an instance of "merge and sort". However, the schemes "multiplication by repeated addition", "division by repeated subtraction", and "exponentiation by repeated multiplication" do seem to be instances of a more abstract scheme, namely "perform one operation by repeating another". Hence schemes can be related to one another in at least two ways: containment and instantiation. What other relationships or interactions among schemes need investigating?

Third, the theory should provide insight on how schemes are "tied" to scheme representations. The classes of features which are important can to some extent be identified from an abstract point of view. What follows in the rest of this paper represents a first step in this direction. A theoretical analysis should look further into these and other issues.

Preliminary Issues of Scheme/Representation Relationships

At least four issues arise when considering how schemes are related to their representations. First, what properties of a representation identify it as representing a particular scheme? I call these the identifying features of a scheme representation. Second, what features do people use to identify schemes from representations? These triggers may not be the same as, or even among, the identifying features. Third, what features are readers conscious of when considering code? These objects of attention may be neither identifying features nor triggers. Finally, how effectively does a particular representation reflect its associated scheme? I call this issue representational fidelity.

Understanding texts involves trying to figure out what the author meant. For program texts, this means determining what schemes the author had in mind. Documentation cannot eliminate this need to "look into the author's head",

since it too contains representations, not schemes. Identifying schemes in program texts requires matching features of the representations to possible schemes. And since programmers make mistakes, the representation may not "reflect" the programmer's intent. When it fails to, two questions arise. First, what does the representation represent? Second, and more basically, what does "understanding the text" now mean? If debuggers use schemes to understand "bad" program texts, there must be some independent criteria which link representations with schemes. These criteria are the identifying features of scheme representations.

But we only need independent criteria when something goes wrong. When reading "normal" or "good" texts, there is reason to believe that people use more direct "cues" to recognize schemes. What is required here is an account of how human readers interact with the various features of a representation, identifying or otherwise. (For more on triggers, see Hassell and Nutter, 1984.)

However, triggers need not always be objects of attention. In a directed prestudy experiment (Hassell and Lind, 1983; Hassell, Lind and Rice, 1983), 40% of subjects asked to identify the lines of pseudo code that constituted a "running sum" loop left out the loop construct itself! Thus it appears that readers may not be aware of essential parts of a scheme, i.e., that they take certain parts so much for granted that they do not spontaneously recall them when asked to do so. But in associated one-on-one studies in which subjects observed debugging code were asked to describe out loud what they were doing, the loop construct played a key triggering role. Thus triggers and objects of awareness need not correspond.

While some issues of understandability lie in the psychological realm, some relate to how "good" particular representations are. Informal evidence abounds that some representations reflect their function relatively clearly, while others obscure it with remarkable success. A few studies have compared the effectiveness of particular modes of representation (see e.g. Sheppard, Kruesi, and Curtis, 1981), but we still know almost nothing about which aspects of representations contribute to representational fidelity.

Conclusion

This paper constitutes a proposal for investigations into the nature and role of programming schemes and their representations. Interest in schemes or some similar construct has emerged from a number of different groups and has already motivated substantial empirical work. But without a clearer notion of what schemes are and of how they are related to their representations, these studies must rest on shaky ground. An improved foundational analysis offers enhanced understanding of the role abstract knowledge plays in program understanding, which can then be exploited both to suggest the role of related abstract knowledge in other kinds of understanding and to provide rich new directions for future research.

References

Adelson, B. Problem solving and the development of abstract categories in programming languages, Memory and Cognition v 9, 1981.

Chase, W.C. and H. Simon. Perception in chess, Cognitive Psychology v 4, 1973.

Curtis, B., S.B. Sheppard, P. Milliman, M.A. Borst, and T. Love. Measuring the psychological complexity of software maintenance tasks with the Halstead and McCabe metrics, IEEE Transactions on Software Engineering v 5, 1979.

Ehrlich, K. and E. Soloway. An empirical investigation of the tacit plan knowledge in programing, Human Factors in Computer Systems, Ablex Inc., 1982.

Hassell, J. and J. Lind. Programming plans as advance organizers and their use in improving programmer debugging performance, AEDS Twenty-First Annual Convention Proceedings, Portland, OR, 1983.

Hassell, J., J. Lind, and J. Rice. Using plan knowledge in debugging: an empirical investigation, Tulane University Technical Report 83-102, 1983.

Hassell, J. and J.T. Nutter. Programming schemes: their role in program understanding and maintenance, AEDS Twenty-Second Annual Convention Proceedings, Washington, DC, 1984.

Magel, K. A theory of small program complexity, ACM SIGPLAN Notices v 17, 1982.

Schank, R.C. and R. Abelson. Scripts, Plans, Goals and Understanding, Lawrence Erlbaum Associates (Hillsdale, NJ) 1977.

Schank, R.C. Remembering and memory organization: an introduction to MOPS. In Strategies for Natural Language Processing, W.G. Lehnert and M.H. Ringle, eds., Lawrence Erlbaum Associates (Hillsdale, NJ) 1982.

Sheppard, S.B., P. Milliman, and T. Love. Human factors experiments on modern coding practices, Computer v 12, 1979.

Sheppard, S.B., E. Kruesi, and B. Curtis. The effects of symbology and spatial arrangement of software specifications in a coding task, Proceedings of the Fifth International Conference on Software Engineering, 1981.

Shneiderman, B. Exploratory experiments in programmer behavior, International Journal of Computer and Information Sciences, 1976.

Soloway, E. and B. Woolf. Problems, plans, and programs, ACM SIGCSE Bulletin v 12, 1981.

Soloway, E., K. Ehrlich, J. Bonar, and J. Greenspan. What do novices know about programming? In B. Shneiderman and A. Badre, eds., Directions in Human-Computer Interaction, Ablex Publishing Co., 1982.

Soloway, E., K. Ehrlich, and E. Gold. Reading a program is like reading a story (well, almost), Proceedings of the Fifth Annual Conference of the Cognitive Science Society, 1983.

Weiser, M. Programmers use slices when debugging, Communications of the ACM v 25, 1982.