Graph Exploration for Software Archeology

Mark A. Foltz
Design Rationale Group, `http://www.ai.mit.edu/projects/drg/`

The Problem: Program comprehension and reverse engineering (i.e., software archeology) remains a major bottleneck for software maintenance. When a programmer must understand a large legacy program whose documentation is scarce, out-of-date, or irrelevant, she must browse its source code to build an effective mental model of its structure and behavior. But source code is far from an ideal representation for this task.

Source code is hard to understand for many reasons: (1) it is broken into many modular pieces, leaving the programmer to reconstruct how they fit together; (2) it contains many symbols whose definitions are located far from their use; (3) it is often browsed in a text editor that lacks context and overviews; and (4) finding where to start browsing in large programs is half the challenge. This research aims to improve on textual browsing for program understanding by developing a graph browsing model of source code that takes into account the cognitive needs and constraints of the programmer.

Motivation: The complexity of large software systems makes it difficult for programmers to learn their structure and organization. Ordinarily, programmers rely on human-generated documentation to obtain this information, but documentation takes time to generate and may be out-of-date. Tools like the one I propose that replace some of the overviews and walkthroughs in missing documentation can help alleviate the productivity penalty of software complexity.

Program comprehension is also a critical part of software maintenance. In an ideal world, instead of expending valuable mental energy searching through and poring over thousands of lines of code, the programmer will be able visually explore the program, literally seeing how it works. If comprehension were less of an impediment to software maintenance, the reality of software evolution could be altered: instead of reimplementing a hard-to-maintain legacy system from scratch, as is often done, designers will find it plausible to adapt and reuse existing source code.

Previous Work: Prior research has contributed a rich palette of software visualizations [4, 1], as well as software diagramming languages for documentation and design [3]. My approach is particularly related to work that is cognitively motivated, i.e. that examines the mental constructs and processes that underlie program comprehension. Work in this area has established that top-down and bottom-up processes both participate in comprehension, as well as the use of coding conventions and domain knowledge [5].

Current Approach and Future Work: My approach is first to collect examples of program understanding behaviors and strategies. A series of observational studies and interviews is underway to obtain such examples. Subjects are asked to browse and understand the source code of an undocumented, unfamiliar program in hypertext form while "thinking out loud" about their comprehension strategy. Interviews allow the subjects to describe browsing strategies and the frustrations they experience when reading other people's code. Initial observations have led to several insights about the cognitive aspects of comprehension, including:

1. Explanations of behavior often proceed from the user interface backward, tracing the execution path in response to a user action (i.e., clicking a button), as in Figure 1.
2. Diagramming is not often used in the comprehension of simple programs.
3. Keyword searching of source code is a desired means of code navigation.
4. Code style matters; an unfamiliar or unintuitive style is distracting.

With more insights like these in hand, I will proceed by modeling the problem of software visualization as graph exploration. Source code can be viewed as a collection of textual fragments linked via a variety of informative interrelationships. The visualization and user interface challenge is to let the programmer explore these fragments in digestible chunks, while giving her useful navigation choices and maintaining her orientation in the overall software. I have proposed a system, DR. JONES, that will incorporate the graph presentation model to permit the orderly and conceptual exploration of an unfamiliar program [2].

Impact: While developing DR. JONES, I expect that many questions about source code browsing will arise, including:

- What is the right level of detail to present the nodes, i.e. the source code fragments?
- Which of the many relationships among code fragments should be highlighted, and which should be suppressed or elided, according to the mode of exploration?
- How much of the program can be displayed at once, subject to the constraints of screen real estate and the user's attentional budget?
- Where should exploration begin, and which navigation choices are appropriate at each step?

Progress on these questions will mean better interfaces for code presentation, code maintenance tools, and integrated development environments. Furthermore, facilitating user exploration of graph-structured systems is a long standing problem in human-computer interaction. Contributions to this area will simplify interaction with virtual models of engineered systems, navigation in hypertexts and knowledge bases, and the exploration of qualitative processes in natural systems (i.e., gene expression).

Research Support: This research is sponsored by the MIT Ford Motor Company Collaboration and MIT Project Oxygen.

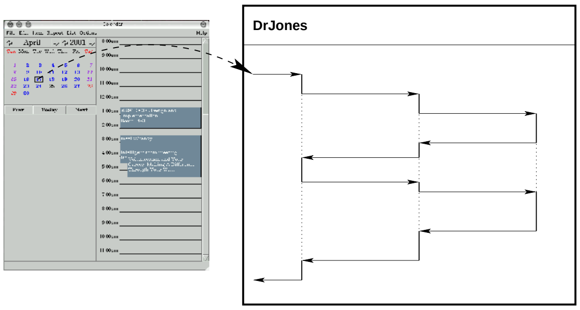

Figure 1: Software "jiggling:" Exploring software by showing the programmer how high level interface behavior maps onto software structure and function.

References:

[1] Stephen Eick. Graphically displaying text. *Journal of Computational and Graphical Statistics*, 3(2):127–142, June 1994.

[2] Mark A. Foltz. Dr. Jones: A software archeologist's magic lens. Doctoral Thesis Proposal, Department of EECS, Massachusetts Institute of Technology, June 2001.

[3] Daniel Jackson. Alloy: A lightweight object modelling notation. Technical Report 797, MIT Laboratory for Computer Science, February 2000.

[4] Wim De Pauw, Doug Kimmelman, and John Vlissides. Visualizing object-oriented software execution. In John Stasko, John Domingue, Marc H. Brown, and Blaine A. Price, editors, *Software Visualization: Programming as a Multimedia Experience*, chapter 22, pages 329–346. The MIT Press, Cambridge, MA, 1998.

[5] M.-A. D. Storey, F. D. Fracchia, and H. A. Müller. Cognitive design elements to support the construction of a mental model during software visualization. In *Proceedings of the 5th International Workshop on Program Comprehension*, pages 17–28, Dearborn, Michigan, May 1997.