How Software Engineering Tools Organize Programmer Behavior During the Task of Data Encapsulation*

ROBERT W. BOWDIDGE
IBM T. J. Watson Research Center, Yorktown Heights, NY
bowdidge@watson.ibm.com

WILLIAM G. GRISWOLD
Department of Computer Science and Engineering, University of California, San Diego, CA

Abstract. Tool-assisted meaning-preserving program restructuring has been proposed to aid the evolution of large software systems. These systems are difficult to modify because relevant information is often widely distributed. We performed an exploratory study to determine how programmers used a restructuring tool interface called the “star diagram” to organize their behavior for the task of encapsulating a data structure. We videotaped six pairs of programmers while they encapsulated and enhanced a data structure in an existing program. Each team used one of three environments: standard UNIX tools, a restructuring tool with textual view of the source code, or a restructuring tool using the star diagram view.

We systematically analyzed the videotape transcripts to derive a model of how the programmers performed encapsulation. Each team opportunistically exploited the features of the tools (e.g., cursors) and the program representation (e.g., ordering of lines in a file) to help them track the current state of the activity. Each method of exploiting structure tracks state in a way that decreases the likelihood of some types of oversights (e.g., missing a required change), but may not address others (e.g., making a change incorrectly), hence requiring a separate check. We also observed that programmers often preferred to design and restructure in an exploratory fashion.

The major challenge of restructuring, then, appears to arise from the fact that it is costly or haphazard to maintain some completeness and consistency properties with the state-maintaining tactics that programmers employ with current tools. The inherent invisibility of some information makes completeness even more costly. These insights have led us to redesign our restructuring tools to better support exploratory design and counter invisibility.

Keywords: restructuring, data encapsulation, empirical study, software tools

1. Introduction

Software maintenance is the greatest contributor to the cost of useful software. Lientz and Swanson found that software maintenance can account for 70% of the total software system’s lifetime cost (Lientz & Swanson, 1980). Boehm cited an Air Force project in which the development cost was $30 per line, but the maintenance cost was $4,000 per line (Boehm, 1975). Much of this cost is attributed to the difficulty of modifying software whose structure has been degraded by the numerous changes that have been layered onto it in the past (Belady & Lehman, 1971). Such changes are necessitated by the need to accommodate the demands of users for new features and changes in the underlying technology. If these changes have not been appropriately anticipated in the system’s design, the change will span
many system modules (Parnas, 1972), incurring high costs and likely degrading software structure (Belady & Lehman, 1971).

One way to lower software maintenance costs, then, is to restructure the system into a more modular form while preserving the original functionality (Opdyke & Johnson, 1990; Griswold, 1991; Opdyke, 1992; Griswold & Notkin, 1993; Johnson & Opdyke, 1993). By isolating the code related to a changing design decision within a module, the change can be applied locally, and hence at lower cost (Parnas, 1972).

Restructuring, however, is a difficult task, requiring a global understanding of the program’s structure as well as global changes to achieve the desired change in structure. If these changes are not made in a complete and consistent manner, the resulting structure may be misleading or the behavior of the program may be inadvertently changed. We developed a prototype tool to assist restructuring (Griswold & Notkin, 1993; Bowdidge & Griswold, 1994), but were unsure whether the technology in this tool should serve as the basis for similar tools designed to restructure large systems. To effectively use the ideas from this prototype to help develop production-quality restructuring tools, we need to understand how programmers use this tool, and how the organization and features of this tool influence how programmers perform maintenance.

To learn how programmers restructure and better understand the problems that they encounter during restructuring, we employed systematic observational techniques on pairs of programmers using one of three tool sets: traditional UNIX editing and searching tools, a prototype restructuring tool with a text-oriented interface, or this same tool augmented with a manipulable graphical visualization—called the star diagram—designed specifically for data encapsulation. The basic question we were asking was “How do programmers use the capabilities of each set of tools to guide their progress in the restructuring task of data encapsulation?” We observed in detail how programmers accomplished a restructuring assignment during a two hour session, and used the resulting videotapes and transcripts to qualitatively characterize how programmers restructure and identify unanticipated issues for future investigations.

The purpose of observing programmers using a variety of tools was not to see which tool set was better. Indeed, our restructuring tools should prove better in certain ways simply because they are specifically designed to ease the task of restructuring, whereas the UNIX tools are not. The restructuring tools are also certainly inferior in other ways because they are prototypes. Rather, we looked at a variety of tools to help us generalize our observations and permit us to make comprehensive improvements to our tools rather than make narrow fixes to the few peculiarities observed in this study.

Although our study was largely exploratory in nature, we did hypothesize that there were two principal problems in program restructuring: making correct global changes to accomplish a structural change, and planning out an entire restructuring activity such as extracting an abstract data type from existing code. We operationalized the first as a set of tasks to carry out a structural change, and the second as a significant planning effort by programmers before undertaking any actual restructuring. In short, we observed the following:

- Our postulated sequence of tasks was essentially correct, but we discovered an additional key task, called the “finding non-literal uses task”, which is concerned with finding