Empirical Software Engineering, 2, 221–267 (1997)
© 1997 Kluwer Academic Publishers, Boston. Manufactured in The Netherlands.

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

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

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

* This work supported in part by NSF Grants CCR-9211002 and CCR-9508745, a Hellman Faculty Fellowship, and an SAIC Summer Fellowship.

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 computations that should be part of a new abstraction, but are not always easily identified with the tools in use. It appears that no tool technology can eliminate the need for this task, but only reduce it.

- Keeping track of the state of the overall restructuring task as well as the state of specific restructuring modifications—what might be called bookkeeping—is a crucial activity. Bookkeeping occurs at many levels in the process: completely performing a specific restructuring change, evaluating progress during a set of changes, and overall sequencing of restructuring activities. Each team exploited structure implicit in the tools (e.g., cursors) and the program representation (e.g., the ordering of lines in a file) to keep track of information regarding the current state of the activity.

  The methods employed by the programmers vary widely, although they share the underlying similarity of trying to achieve certain properties of completeness and consistency of a change. Each method of exploiting structure decreases the possibility of some class of oversights (e.g., missing a required change), but does not address others (e.g., formulating a flawed design), hence requiring additional integrity checks. In general, these tactics amount to maintaining "to-do lists" of data or design considerations that have yet to be processed. Although we did not anticipate the importance of bookkeeping, it indirectly supports our hypotheses about the challenges of making correct global changes and planning to carry out a restructuring.

- The nature of the planning effort varied widely amongst teams, with some teams choosing a genuinely exploratory style. This variation appears to be due to the interplay between tool design, a complex design task, and personal preference. Tool design influences what information can be held for the programmer. If a design task is sufficiently complex, the programmer may not be able (or want to invest the time) to completely understand all the issues in a vast design space. Some programmers, however, collected more-or-less complete information.

Together, these discoveries led to the following insights about restructuring:

- Due to the large amount of distributed information that must be managed during restructuring, programmers use the tools available in a way that attempts to reduce the need to mentally recall information. In particular, when faced with a complex design task, a programmer will use tool features in a way that allows the tool to store information conveniently for the programmer. Moreover, if the tool cannot conveniently store the information, the programmer will order subtasks in a way so that the information is immediately used and can then be forgotten.

- There is an inherent invisibility (Brooks, 1987) of design information in restructuring. For example, it is not always straightforward to identify all the computations in a program that correspond to a proposed function. It is this invisibility that results in the finding non-literal uses task and encourages exploratory behavior as an alternative to a costly up-front analysis.

Prior to this study, we had thought of the star diagram as a tool for visualizing structure in a program and providing functionality to act on that visualization and the underlying program.

```
(get-line *line-storage* lineno)
...
(get-line *line-storage* lineno)
...
(define get-line
  (lambda (ls line)
    (list-ref ls line)))
```

(a) Before Inline Parameter

```
(get-line lineno)
...
(get-line i)
...
(define get-line
  (lambda (line)
    (list-ref *line-storage* line)))
```

(b) After Inline Parameter

Figure 1. The Inline Parameter transformation takes a variable or expression that occurs as a parameter to a function call, destroys the parameter, and replaces the function's argument with the expression. This transformation only works if all calls to the function have matching arguments. Note that this transformation converts the code so that the programmer directly sees that `*line-storage*` is acted upon by the `list-ref` function. (When programming in Scheme or Lisp, asterisks around a variable name are used as a convention for indicating global variables.)

We had not considered the star diagram as a mechanism for helping the programmer to orchestrate a complicated design activity. However, our observations and insights led us to give explicit consideration to the unanticipated affordances[1] of each tool set that programmers used to maintain state during modifications. In particular, we made changes to improve the star diagram's ability to store task-related state and to allow the programmer to engage in exploratory design. We have also taken steps to increase both the visibility of program details relevant to design and the programmer's awareness of invisibility issues.

The remainder of this paper describes these results and briefly discusses other insights into the design of the star diagram. Additionally, this paper documents the observational techniques we employed so that others may use them. Although time-consuming, they require few other resources and are well-suited to studies of research prototypes.

2. Background: Meaning-Preserving Restructuring Tools

Our basic restructuring tool allows a program's structure to be changed without affecting its behavior, thus ensuring that the restructuring does not introduce bugs into the program (Griswold, 1991; Griswold & Notkin, 1993). To use the restructuring tool, the programmer selects meaning-preserving restructuring transformations to apply to selected portions of the code. Each transformation changes the selected code, as well as other related portions of the program, to achieve the restructuring. There are twenty basic transformations, including inlining a parameter (See Figure 1), changing order of statements, replacing a function call with the body of the function, or replacing a set of similar statements with calls to a new function (See Figure 2), and their inverses. If the tool detects that the programmer's transformations could change the running behavior of the code, the tool prohibits the change and highlights the limiting dependencies. The current restructuring tool operates on the imperative language Scheme,[2] and correctly manipulates programs that use pointers and assignment.

```
...
(let
  ((len (length (list-ref *line-storage* lno))))
...
  )
...
(set! linelen (length (list-ref *line-storage* linenum)))
...
```

(a) Before Extract Function

```
(define words-on-line
  (lambda (line-number)
    (length (list-ref *line-storage* lno))))
  ...
  (let ((len (words-on-line lno)))))
  ...
    )
  ...
  (set! linelen (words-on-line linenum))
  ...
```

(b) After Extract Function

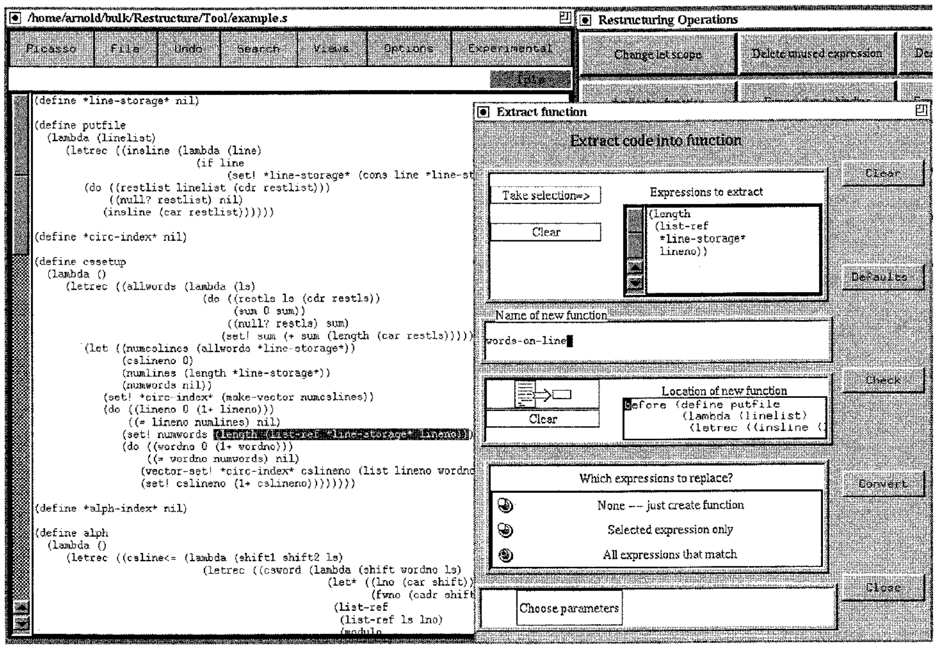
Figure 2. The Extract Function transformation takes a common expression occurring throughout the code, creates a new function whose body is the expression, and replaces occurrences of the expression with calls to the new function. The tool user also specifies the name of the new function, its location in the code, and what parts of the expression will be parameterized.

The star diagram's user interface has two views with different styles of interaction, the "text-based" restructuring tool and "star diagram".

2.1. Text Interface for the Restructuring Tool

The "text-based" restructuring tool displays the source code of the program in a scrolling window. Transformations are selected from a panel, which pops up a dialog that the programmer fills in by typing or with selections from the source code (See Figure 3). The interface provides few searching features, so programmers must use existing UNIX tools to perform some queries.

To extract a new function from existing code, the programmer selects the Extract Function button from the Restructuring Operations menu, partially covered in the upper right of Figure 3. The Extract Function transformation panel, on the right in the figure, specifies the parameters required to perform the transformation.



```
(define *line-storage* nil)

(define putfile
  (lambda (linelist)
     (letrec ((insline (lambda (line)
                          (if line
                               (set! *line-storage* (cons line *line-st
        (do ((restlist linelist (cdr restlist)))
            ((null? restlist) nil)
           (insline (car restlist))))))

(define *circ-index* nil)

(define cssetup
  (lambda ()
     (letrec ((allwords (lambda (ls)
                          (do ((restls ls (cdr restls))
                               (sum 0 sum))
                              ((null? restls) sum)
                             (set! sum (+ sum (length (car restls))))))
        (let ((numcslines (allwords *line-storage*))
              (cslineno 0)
              (numlines (length *line-storage*))
              (numwords nil))
           (set! *circ-index* (make-vector numcslines))
           (do ((lineno 0 (1+ lineno)))
               ((= lineno numlines) nil)
              (set! numwords (length (list-ref *line-storage* lineno)))
              (do ((wordno 0 (1+ wordno)))
                  ((= wordno numwords) nil)
                 (vector-set! *circ-index* cslineno (list lineno wordno
                 (set! cslineno (1+ cslineno))))))))

(define *alph-index* nil)

(define alph
  (lambda ()
     (letrec ((csline<= (lambda (shift1 shift2 ls)
                          (letrec ((csword (lambda (shift wordno ls)
                                              (let* ((lno (car shift))
                                                     (fwno (cadr shift
                                               (list-ref
                                                (list-ref ls lno)
                                                (modulo
```

```
(length
 (list-ref
  *line-storage*
  lineno))
```

```
words-on-line
```

```
before (define putfile
         (lambda (linelist)
          (letrec ((insline (
```

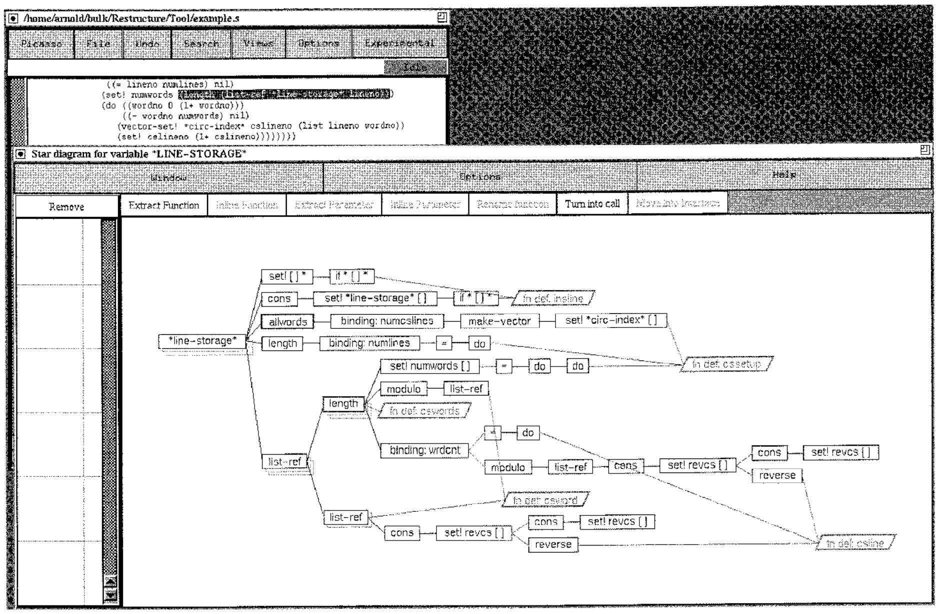
Figure 3. User interface for the text-based restructuring tool, in the process of performing Extract Function.

2.2. The Star Diagram: Supporting Encapsulation

Although the text-based restructuring tool solves the problem of making complicated behavior-preserving global changes to a program, the programmer is still faced with the problem of understanding the program's structural problems. Such an understanding is complicated by a straightforward display of the program text—as shown in Figure 3—which does not convey how an expression is related to other expressions elsewhere in the program in terms of redundancy or the use of related values. Rather, the programmer is shown a significant amount of local computational context, only a small part of which may be relevant to the current structural problem.

The star diagram graphical interface (see Figure 4) helps with one important instance of this problem: performing data encapsulations. A star diagram graphically and compactly presents only those computations in the program that use a chosen data structure, helping the programmer to select and create the functions to completely encapsulate it (Bowdidge & Griswold, 1994). These functions and the underlying data structures collectively represent an abstraction or module that hides the data structures from the rest of the system.

The root of the star diagram, on the left, represents all uses of a variable being encapsulated; nodes at the first level of the diagram indicate operations directly using the variable; nodes at the second level represent operations nested around or consuming the result of the previous



```
((= lineno numlines) nil)
(set! numwords (length (list-ref *line-storage* lineno)))
(do ((wordno 0 (1+ wordno)))
    ((= wordno numwords) nil)
  (vector-set! *circ-index* cslineno (list lineno wordno))
  (set! cslineno (1+ cslineno)))))))
```

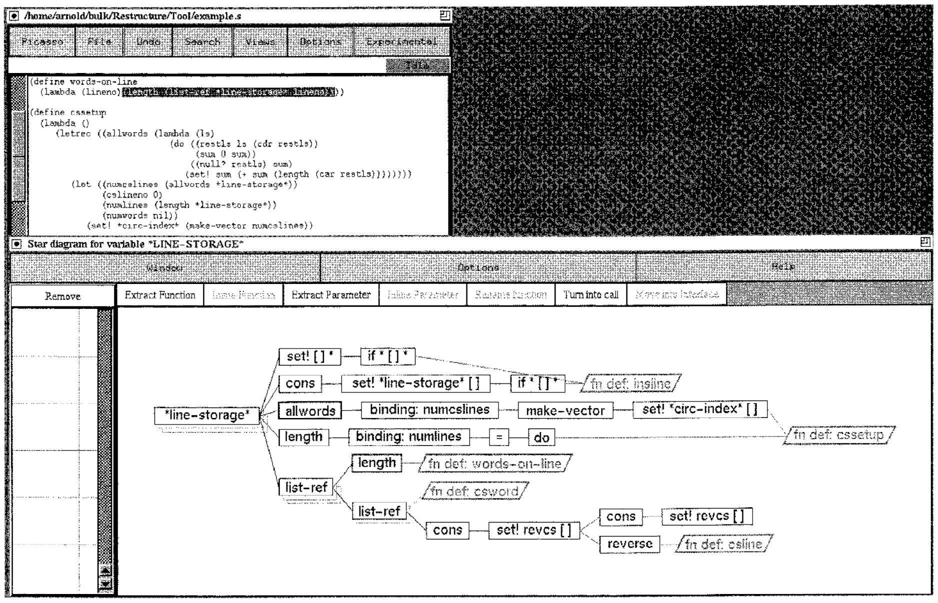
Figure 4. Star diagram visualization for restructuring to encapsulate data structures. This particular view shows the program after several applications of Inline Parameter. The boxes to the far left of the star diagram window will list the names of functions representing the interface to the abstraction being created.

operation, and so forth. Thus, each expression or statement containing a use of the data structure is shown as a chain of operation nodes representing the operations performed on the use. Each chain is terminated at the function definition in which the use occurs—displayed as a parallelogram—-thus showing the context of each chain.

When nodes with similar labels connect to the same parent node, the matching nodes are combined into a single node. An icon representing a “stack” of nodes denotes that the node actually corresponds to several similar expressions. Stacking combines similar expressions from possibly separate scopes. Graphically overlaying similar expressions means that each unique computation is represented just once, helping the programmer to identify new operations and to manipulate similar expressions together to create a new operation. A node near the left of the tree is an expression or statement that might implement a “low level” operation, encapsulating little more than the data structure representation itself. A node farther to the right might implement a “high level” operation.

Figure 4 presents a star diagram during the encapsulation of the `*line-storage*` variable in the program used for this study. The expressions corresponding to the `words-on-line` abstraction shown in Figure 2 are presented by the left-to-right chain of `*line-storage*`, `list-ref`, `length`, near the bottom of the diagram.

Each node in a star diagram is linked to a text view of the source so that elided details are readily accessible. Selecting a node displays and highlights its corresponding source code in



```
(define words-on-line
  (lambda (lineno) (length (list-ref *line-storage* lineno))))

(define cssetup
  (lambda ()
    (letrec ((allwords (lambda (ls)
                         (do ((restls ls (cdr restls))
                              (sum 0 sum))
                             ((null? restls) sum)
                           (set! sum (+ sum (length (car restls))))))))
      (let ((numcslines (allwords *line-storage*))
            (cslineno 0)
            (numlines (length *line-storage*))
            (numwords nil))
        (set! *circ-index* (make-vector numcslines))
```

Figure 5. After the `words-on-line` function has been extracted by the Extract Function transformation.

the program text view. In the case of a “stacked” node—where there are multiple statements associated with a node—the programmer can navigate to each occurrence through a menu selection.

A program is restructured with the star diagram by selecting a node in the diagram, and pressing a restructuring transformation button. Dialog boxes prompt for any additional information. The star diagram provides transformations tailored to encapsulation. Transformations outside the encapsulation paradigm can be accessed from the text view of the restructuring tool.

The star diagram provides additional assistance for the encapsulation task. The functions encapsulating the data structure are recorded in a list located on the far left side of the star diagram window. When the programmer creates a new function appropriate for the interface of the new module, the programmer adds the function to the interface by selecting the function node in the star diagram and pressing the Move into Interface transformation button. This operation adds the function name to the interface list, then removes the relevant calculations from the star diagram. (The operation can be reversed by selecting a function and pressing the Remove button at the top of the list.) Through this process, the star diagram representation clearly distinguishes between uses of the data structure that are unencapsulated (those in the star diagram) and those that are encapsulated (those occurring within the functions in the interface list.)

Transformation Extract Function is a fundamental star diagram transformation. It is invoked from the star diagram by selecting a node stack and then selecting the Extract

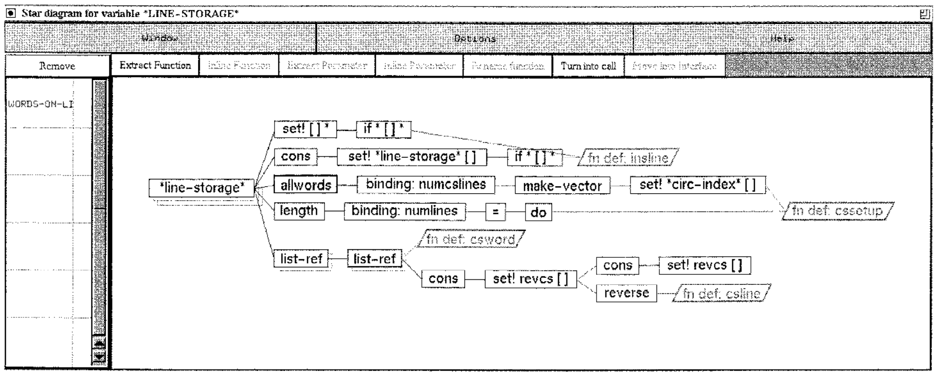


Figure 6. After moving the new function from the star diagram into the module interface with the Move into Interface operation.

Function button (Figure 4). A dialog appears with a summary of each computation associated with the node stack, and the programmer selects those which are to be replaced by a call to the new function. The tool then presents a dialog, similar to that shown in Figure 3, for choosing the function's name and what parts of the computation will be its arguments. Finally, the tool verifies that the chosen computations are in fact behaviorally identical and that the resulting transformation will preserve the functionality of the program. If so, the tool performs the transformation and updates the star diagram to reflect the changes to the program (Figure 5). The programmer can then click on the new function definition in the diagram, and click the Move into Interface button, removing the definition and its body from the star diagram and placing it in the interface panel to the left (Figure 6). Repeating this basic sequence—interleaved with other transformations that prepare the code for function extraction—results in an empty star diagram, signaling that every computation on the data structure has been abstracted as a module operation.

3. Study Method

3.1. *Motivation for Using Systematic Observational Techniques*

Choosing a method to evaluate our tools was not easy. We know of only one study of programmers performing restructuring of any kind (Griswold & Notkin, 1992), and it was ad hoc and focused on the mechanics of the change. Also, our restructuring tool is a prototype that can only be used on small Scheme programs, so a case study in an industrial setting is currently infeasible. Because little is known about how programmers use these tools, it is not possible to isolate and test a few experimental variables. Indeed, Schneiderman and others have noted that understanding the context of usage is crucial to understanding how software tools are used (Schneiderman & Carroll, 1988), and so unless we understand the full context of restructuring and encapsulation tasks, we may make inappropriate decisions in the design of a restructuring tool. Although an anthropological approach of observation in a completely natural setting would be ideal (Blomberg et al., 1993), our desire to understand the use of prototype tools and to compare different tools in similar settings precluded such an approach for the time being.

We can understand how programmers perform restructuring by observing their behavior during a restructuring task. These techniques, most generally called systematic observational techniques (Weick, 1968), are common methods for studying real-world behavior in the social sciences. When the observations are recorded on video or audio tape and then analyzed, these techniques can be referred to as exploratory sequential data analysis (Sanderson & Fisher, 1994). When applied to understanding how programmers work, approaches can range from using analysis of video and transcripts to test a specific research hypothesis (Gray & Anderson, 1987; Ericsson & Simon, 1993), to using verbal data for exploratory understanding of planning, behavior, and problem solving (Curtis et al., 1988; Guindon, 1990b).

Exploratory studies can be divided into observations in the workplace and observations in a laboratory setting. Exploratory studies involving cognitive or problem-solving issues are often performed in a laboratory setting. The setting allows the experimenter to choose tasks and programs designed to expose specific behaviors, and to record observations easily. Letovsky observed programmers' program comprehension methods (Letovsky & Soloway, 1985) and browsing strategies (Letovsky, 1986). Sutcliffe and Maiden explored mental behavior of analysts during problem solving (Sutcliffe & Maiden, 1992). Cousin used observations of programmers to identify information that should be provided in a software engineering environment (Cousin & Collofello, 1992). Lange and Rosson both documented reuse strategies in an object-oriented programming environment (Lange & Moher, 1989; Rosson & Carroll, 1993). Other studies tested the design of a database-style programming environment for the Dylan language (Dumas & Parsons, 1995) and listed common problems in Macintosh programming environments (Houde & Sellman, 1994).

We chose to systematically observe programmers as they performed restructuring during a two hour restructuring exercise. With such methods, we can analyze the behavior of a small number of subjects in a simulated setting, and produce qualitative descriptions rather than quantitative results. Inferences from such descriptions can be used as guidelines for tool development or provide hypotheses that can be tested with a specific experiment at a later time.

Our study is modeled on Flor's studies of organization within groups of programmers in a laboratory setting (Flor & Hutchins, 1991). Flor's work differs from the other programmer studies because he used pairs of programmers working together as subjects. This technique, known as *constructive interaction* (Miyake, 1986; Wildman, 1995), uses the programmers' dialogue to observe their problem solving processes in a more natural context than single-person think-aloud methods. Because the programmers are also working in a familiar work environment, their dialogue and actions on the computer may also reflect habits and patterns typical of the programming culture. By observing the two programmers and the computer together, we can also identify what information programmers must record or examine to perform the restructuring.

3.2. Setup

To determine a suitable setup for the studies, we performed a pilot study on the star diagram using six programmers; another six participated in a previous study enhancing a C program. These studies revealed that a small, focused task was necessary in order to prevent programmers from getting onto time-consuming tangents. We also found that some programmers were unwilling (as opposed to unable) to produce a program with a different structure without sufficient motivation or explicit directions.

For the final setup, we chose to instruct teams of two programmers[3] to perform a specific enhancement to a somewhat complex 150 line Scheme program by using a "restructure, then enhance" process. We ran the sessions in a laboratory setting to limit interruptions and facilitate video recording for later analysis. Teams worked at a single monitor and were told that they had two hours to work on the task, although they could continue up to an additional hour if they chose. (We allowed the programmers to continue because we assumed most would strongly desire to finish the task during the session. In addition, because we were interested in behavior, not speed of modifications, we had little reason to force programmers to complete the task in an arbitrary time period.) The task was limited to restructuring a small program because larger programs could not be restructured using the current restructuring tools, and because detailed exploratory analysis of the resulting video data would have been infeasible. Three teams finished the complete task in the allotted time, two finished the encapsulation but did not have time to add the enhancement, and one team did not finish the encapsulation.

3.2.1. Conditions

Each team performed the restructuring task using one of three tool sets: using standard UNIX tools, using a text-based restructuring tool, and using the star diagram interface. Although the star diagram and restructuring tool teams primarily used the tools they were given, some tasks not supported by the restructuring tool led them to use standard UNIX tools. For example, because the restructuring tools omit comments in their presentation of the source code, programmers used standard editors to view the comments in the original source code. Allowing the programmers to use UNIX tools could encourage them to ignore the new tools, but the programmers were explicitly told at the beginning of the experiment and in the instructions that they were "encouraged to first see if the restructuring tools could help them perform the task."

Although most teams accomplished major parts of the task with the suggested tools, one team using the text-based restructuring tool stopped using the restructuring tool halfway through the session because of frequent crashes, and instead finished the task using standard UNIX tools. Their behavior is included in this study because their methods for modifying the file with an editor were distinct from methods used by other teams, and thus highlights the range of techniques programmers might apply when restructuring.

Table 1. Years of industry experience, UNIX experience, and Lisp experience (including classroom experience) for subjects, grouped by team and tools given. * Text restructuring team 1 stopped using the restructuring tool due to frequent crashes, and instead finished the encapsulation with UNIX tools.

Condition, team	**Subject**	**Experience (years)**			**Progress in task**
		Industry	**UNIX**	**Lisp**	
UNIX tools 1	F	1	5	1	performed encapsulation
	B	0	6	5	only
UNIX tools 2	P	7	7	10	encapsulation and
	C	1	6	3	enhancement
text restr 1*	D	7	7	6	encapsulation and
	M	0	6	2	enhancement
text restr 2	T	0	9	4.5	encapsulation and
	A	4	5	3	enhancement
star diagram 1	J	10	9	2	performed encapsulation
	K	2.5	6	2	only
star diagram 2	I	0	4	5	encapsulated 5 of
	R	16	20	4	6 functions
avg. / std. dev.		4.0/5.1	7.5/4.2	4.0/2.5	

3.2.2. Selection of Subjects

There were a total of twelve programmers working in six teams (See Table 1). Subjects were either graduate students (many with industry experience), or programmers from local industry. Subjects were chosen because of their experience in programming and knowledge of Lisp-like languages, ensuring that programmers would not face difficulties because of the programming language used. All programmers were familiar with the UNIX programming environment, so they understood the tools they had available and how to use them. All understood the concept of modularization and had experience programming in an object-oriented language, thus ensuring they were familiar with the basis of the encapsulation task. Although this group may be somewhat unusual in its characteristics, we expect that large-scale restructuring is an activity most often carried out or directed by system architects, who are highly trained specialists themselves.

Programmers were paired according to when they were available and assigned to conditions according to when the restructuring tools were ready for the experiment. Each programmer received a payment of $5/hour for participating. Although the money was an inducement to participate, we found all the programmers to be extremely motivated, many working beyond the nominal two-hour time limit.

3.2.3. *The Task, Process, Program, and Instructions*

Subjects were first informed about what would occur during the session, and were asked to sign standard consent forms informing them, for example, that they could stop participating at any point, and that at any time in the future they could ask for the videotapes and logs to be destroyed. Programmers then received printed copies of the task instructions (Appendix A), a definition and example of encapsulation and, if applicable, a 15 minute demonstration of a restructuring tool and instructions on its use. Programmers had two hours to perform the task, although they were free to continue up to three hours if they felt close to finishing the task, and wanted to continue. Programmers were then given a questionnaire to identify their background and were debriefed in an open-ended interview regarding what they did and why they did it. Interviews usually lasted about twenty minutes.

The programmers were given an implementation of the KWIC indexing program (Parnas, 1972), written in a functional decomposition style (Appendix B). Although short, KWIC is not a "toy" program. The program is about 150 lines, containing 14 functions and four major global variables used throughout the program. The program also contains nested functions.

The task given to the programmers was to change the internal representation of the main data structure from a "list of lines" representation to a "list of words" representation with an auxiliary data structure identifying line breaks. The modification requires examining all functions of the program and performing several global changes. Programmers were asked to first encapsulate the data structure storing the internal representation of the file being indexed (an array pointed to by the `*line-storage*` variable), creating a new module that hid the `*line-storage*` data structure behind a set of functions which acted as the interface to the module. The encapsulation was not to change the program's running behavior. Teams were asked to next perform the enhancement. By enforcing this two-phase process, the programmers were more likely to perform a separate activity that could be identified as data encapsulation.

3.2.4. *Setting*

The laboratory was set up for recording and observation to be as unobtrusive and realistic to the programmers as possible. The laboratory is in fact a workspace for programmers. They had access to paper and pencil for note-taking and sketching, which was collected for later analysis. Only the subjects and experimenter were present in the laboratory. The experimenter was normally out of the subjects' line of sight, and was present only in case the restructuring tool crashed. A video camera was also set up behind the programmers, out of their line of sight. Clip-on microphones were used for audio capture. We found that the subjects quickly forgot about the presence of the camera, microphones, and experimenter. In particular, the subjects would sometimes block the camera's view of the screen, accidentally brush the microphone (without comment), and spontaneously digress onto topics unrelated to the study. Still, the experimental setting differs from a work environment in the fixed time limit, lack of access to printouts, paired work, and size of the program. We discuss these four influences in turn.

Although programmers in industry would not be asked to restructure a program in two hours, it is common for software design problems to be solved under extreme time constraints (Guindon, 1990b). Moreover, the programmers were able to complete the task in the given time.

Programmers in industry would also have access to additional methods of bookkeeping, such as marking printouts of the code. Programmers in the study were not given printed listings in order to force them to browse the code on the computer and in sight of the video camera, making analysis easier. Because the size of the program made substantive note-taking on paper copies unnecessary, the lack of printouts for note-taking should not affect their behavior significantly.

Programmers may not work closely with others on a daily basis. However, programmers do work closely on occasion, and the experimental setup mimics such interactions. Some programmers noted that the greatest change between working alone and working with another person in this setting was that the process of working through solutions with another person caused them to discuss and avoid unproductive actions before actually performing the action. Thus, the programmers made fewer observable errors than if they were working alone, but we were still able to identify their misconceptions from their dialogue. Because our study focused on capturing overall behaviors, rather than error rates or time spent in erroneous behavior, the effect of the team organization weakly influenced our observations.

The size of the program was smaller than typical programs; however, the code was complex enough to limit the programmers' abilities to memorize every detail of the program. The fact that all groups either introduced errors or encountered problems with planning is also indicative of the adequate complexity of the task. On larger systems, we would expect to see such issues only magnified. Because the Scheme restructuring tool could only handle small programs, we could not study scalability issues for the star diagram. However, separate studies indicate that improvements are required for the star diagram to scale (Bowdidge, 1995; Griswold et al., 1996).

3.3. Recording Method

In order to record the sessions for later analysis, we used videotape to record programmer discussions and gestures, and used keystroke logs for computer actions. To facilitate audio analysis, we used two clip-on microphones recording to separate audio tracks of the videotape. The two microphones improved sound quality over a fixed microphone, and enabled us to distinguish between speakers more readily. We recorded keystrokes and mouse actions using the UNIX `script` command, as well as logging within the restructuring tool. We videotaped the screen to observe pointing motions, identify programmer's focus of attention, and to synchronize the keystroke logs, actions, and dialogue. Wall-clock time was imprinted on the videotape. Notes written by the programmers were saved. Because the video camera was usually focused on the screen, we identified when programmers wrote notes by context of their discussions or writing noises.

3.4. Analysis Method

To understand how the different tools affected how the programmers restructured, we first recorded the data, identified relevant issues, condensed the data by coding (reducing the data by categorizing episodes according to a small set of concepts), then compared the reduced accounts (Chi, 1997; Strauss, 1987). We focused our analysis on how programmers restructured, but beyond that narrow question, our hypotheses were built up as we noted interesting patterns in the behavior of one team and asked questions about whether the patterns held in other sessions.

The first step of analysis was to transcribe each videotape into a written verbal account, first beginning with a rough transcript for the session, then creating a transcript of all conversation and actions. (See Appendix C for a sample transcript.) We found we could not pick and choose what to fully transcribe because of the nature of exploratory studies, and because the interesting decisions in restructuring seem to be distributed throughout a session. We found it took about thirty to forty hours to create a full transcript of each two hour session with speakers identified and actions noted. We tried to get verbatim accounts of what the programmers said so the transcripts could be our primary tool for studying each session. All the transcripts contained frequent references to the wall-clock time embedded in the videotape so we could easily find a given section of tape when we needed more context. Linking keystroke transcripts to the text was more difficult because of the lack of timing information, but we identified rough actions in the transcript (actions on the restructuring tool interface, commands to the UNIX shell, etc, transformations performed). We continued to fill in and refine the computer actions as we found the need for order and timing of actions.

To identify relevant issues, we first understood the entire tape by creating rough transcripts from a single watching of the tape and from our transcribing of the activity logs. (We tried to have others transcribe tapes, but we found that the act of transcribing helped our analysis activities.) We then cataloged our observations using colored index cards, with each card representing a key issue or interesting behavior, and each color represented a category of observation. Each card named a specific event, behavior, or hypothesis, then a list of instances of the behavior, arranged by team and location on the videotape. Our categories included:

- interesting behavior or events that occurred for one team (e.g., J and K visited nodes from top to bottom in the star diagram view);
- tabulation of the behavior of all teams for a given question (e.g., when did each team run the program?);
- hypotheses, issues, inferences (e.g., presentation order affects programmer behavior).

Because we were performing bottom-up analysis, our usual procedure was to represent an unconnected fact as a unique issue, then create tabulation cards as specific questions or patterns emerged. The cards served as a written record of likely issues to study, permitted sorting in multiple ways to identify trends, supported marking, and in general provided the flexibility needed to explore the data.

As we proceeded, patterns of programmer behavior emerged, such as the order in which programmers visited or changed functions. When we decided a pattern represented an important research question, we coded the data to reduce the transcript to an account only focused on the pattern being examined. Coding usually involved finding all instances of a given behavior, then excerpting the time, context, or duration of the behavior. When we studied the reasons why programmers switched between the star diagram view and text view, we coded by marking the transcript at such points and tabulating the cause for switching views. (This issue proved uninteresting.) Coding for how programmers described restructuring transformations involved listing the metaphors used. Comparing when programmers visited each function involved creating a time-line showing the time spent examining each function. For understanding overall behavior, we noted the overall process the programmers followed: when they visited the top of each file, when they created a new function, when they converted a call, etc. This data could then be converted into pictures to explain a concept and match to a idea (e.g., see Figures 8, 10, and 11).

When coding for infrequent events, we searched the transcripts for similar or divergent behavior. For frequent events, we segmented or coded specific instances, then produced an outline of behavior based only on that action. As an example of the former, to understand how the teams evaluated their progress, we first discovered the existence of an explicit test for completeness in one transcript. We then checked the other transcripts to see how the other teams behaved with regard to completing a subtask.

We used the latter method for reducing the data to analyze the phases a team went through. One coding kept track of what procedures the programmers were examining at what times. The UNIX tools teams usually passed through the text linearly with a few forward or backward passes. The star diagram teams usually scanned down the star diagram, then navigated to the text at each node. A second coding identified dialogue that could signify a change of focus or intent:[4]

I: [I and R have been examining uses of `*line-storage*`] Let's try to remember what our goal was for a second.

R: Well, I think the first thing was we're supposed to encapsulate these global variables, right?

We then produced a time-line and outline describing the overall behavior of each team. We used the outline of actions to identify how programmers maintained state within transformations and identify the visitation strategies the programmers used.

In general, we found codings related to concrete issues such as process and task ordering more valuable than more abstract codings, such as for motivation. Part of the problem is that it is easier to measure and identify ordering than extract reasons why a programmer performed a given action. Because we are not—as in a think-aloud study—asking for justification of actions, a programmer will only expose motivations when dictated by the circumstances of the collaborative work. For example, we could not easily answer the question, "Why did programmers navigate from the star diagram to the text?" with our data.

One author performed the analysis, and the other evaluated the hypotheses against the video data. We also discussed our observations and hypotheses with some of the subjects.

Because of the exploratory nature of our study, we used only a single coder and performed no inter-coder reliability tests.

4. Observations and Model of the Encapsulation Process

4.1. Model

Because of our interest in identifying the difficulties that teams might encounter during restructuring, we postulated a set of five kinds of tasks that we expected a team to perform during restructuring. Observing how each team realized these tasks helped us understand how the teams used the tools, and exposed the similarities between the different teams in spite of the differing tools. In the process, we discovered a sixth task, called the finding non-literal uses task. All six activities are summarized in Table 2. By formulating a model and then testing its usefulness against the transcripts, we avoided, as best we could, either deriving a model that did not correspond to our broader experience or imposing our preconceptions about what was happening.

To make the model as descriptive as possible, we chose to derive a narrow model of encapsulation by restructuring, rather than of maintenance in general (Collofello & Bortman, 1986). Consequently, not all programmer activities fit within our six categories. For instance, the teams in this study frequently engaged in a general program understanding activity when first presented with the program, and later ran and debugged the modified program. To check that our model was representative, we discussed our model with some of the programmers. They found it to be representative of what they did, although they noted the omission of the early program understanding activity in our model.

Two of the more difficult tasks for this study were the finding non-literal uses task and the choosing functions task, so we describe them in more detail.

4.1.1. The Finding Non-literal Uses Task

Because this implementation of KWIC is designed using a functional decomposition, `*line-storage*` appears literally only at the top of the program's calling hierarchy (as arguments to functions). As a result, some programmers decided to localize the uses of `*line-storage*` so that the variable is not passed down through all procedures, but used directly in the computations. Finding, and in some cases inlining, these indirect uses constitute the finding non-literal uses task. Figure 1 provides an example of this task.

4.1.2. The Choosing Functions Task

In KWIC, encapsulating the `*line-storage*` data structure requires choosing a set of functions to act as the interface to the new abstraction. The programmers' choice of functions roughly corresponded to two major abstractions. Some teams chose to abstract away only the "lines" representation by creating the functions `number-of-lines`, `get-line`,

Table 2. Model for encapsulation. The model does not specify temporal ordering because programmer behavior varied significantly between teams.

Task	Subtask
Finding variables	Navigate to uses of the data structure being encapsulated. View the relevant code.
Finding non-literal uses	Restructure the code to expose uses OR follow dataflow to identify copies of the data structure.
Grouping uses	Match similar expressions. Identify abstract operations.
Choosing functions to create	
Creating functions	Create definition. Unit test the new functions. Convert common expressions into calls to new functions. Test that all expressions have been converted to calls.
Detecting completion	Test done with transformations. Test functionality of system.

and `insert-line`. Others chose to abstract away the "words" representation as well by creating other functions such as `get-word-on-line`, `number-of-words-in-file`, and `length-of-line`. Both approaches adequately encapsulate the representation for the subsequent change.

Programmers had two methods for realizing the functions that encapsulated the data structure. In some cases, existing functions such as `insline` (which inserts a line into the line storage data structure) and `allwords` could be used as-is in the new encapsulation. In other cases, the programmer could find likely functions by identifying common expressions that should be converted into functions. Calls to the new functions could replace several expressions throughout the code that directly access the data structure. The finding non-literal uses task can affect the choice of functions because not finding uses hides some computations on the representation.

4.2. High-level Observations of Each Team

Our task model provides a framework for describing how the programmers completed the restructuring task. By describing behavior relevant to this model, we also provide context for later discussions of specific bookkeeping methods used by each team. The following describes the behavior of each team in greater detail.

4.2.1. Behavior of F and B (UNIX Tools Team 1)

F and B used `emacs` and `more` to view the code. They read linearly through the code multiple times, each time focusing on a different issue. While first reading the code, they focused on understanding the code as a whole, and saw the uses of `*line-storage*` and `ls` in passing (i.e., the finding variables task and finding non-literal uses task). To choose the functions, they re-examined the code (again with a single pass through the source code), matched the common expressions by recall (the grouping uses task), chose operations that matched the abstraction represented by the current data structure, and wrote down the names of the new functions to create. They chose a word-oriented abstraction for the module to be created. F and B then created the new functions, and made another pass through the code to convert the expressions, replacing the expressions that matched the new function bodies with calls to those functions. The programmers incorrectly remembered the order of parameters for certain functions when they created the function calls, and spent a fair amount of time trying to understand why the code no longer worked. Because of a design mismatch between their new functions and the existing code, the new functions did not easily replace the code in the `allwords` function. As a result, they recoded `allwords` using a different algorithm.

F and B completed the restructuring task and tested the restructured code, but they ran out of time before they could perform the enhancement.

4.2.2. Behavior of P and C (UNIX Tools Team 2)

P and C performed the requested restructuring and enhancement with the `emacs` editor. They first looked through the source code to understand it, then made a second pass to find all the uses of the `*line-storage*` data structure. They noted that `ls` was usually the name for `*line-storage*` when it was passed to other functions, and so examined uses of `ls` in calls to built-in operations as possible uses of `*line-storage*`. As they found uses of `*line-storage*`, they wrote down the name of the enclosing function to be used as a search tag later to revisit the uses. After understanding the code and identifying the uses of `*line-storage*` in passing, they then discussed their observations. When they began changing the code, they first created empty function declarations for the functions they decided that they needed. They then filled in the bodies of the new functions with code meeting the needs of the enhancement, and tested the functions alone to make sure they behaved correctly.[5] Next, they returned to all the uses and inserted the calls on the appropriate functions, using the list of functions using `*line-storage*` or `ls` to navigate to the next use in the file. They found during this process that they incompletely examined the `allwords` function. P and C originally thought that they could slightly modify `allwords` to call `get-line` and `size-of-line`. When they went to change the code, they examined it again and found that it represented the number of words in the file, a function they had already written as part of their new interface without knowing that it was a function they actually needed. When they finished adding the calls to the new functions, they tried to load the code into Scheme. After fixing a number of typos, the code loaded and ran correctly. They assumed that since they had applied the changes to all the places they thought necessary, they must be finished, and so no explicit completion test was performed.

4.2.3. Behavior of D and M (Text-Based Restructuring Tool Team 1)

D and M initially were given the text-based restructuring tool. They used `vi` and the restructuring tool's source code window to understand the code. They then used the text-based restructuring tool to start the encapsulation. They first tried to simplify browsing and understanding the code by using the Move transformation to reorganize functions in the code. Next, they began the choosing functions task by creating a function around the get-line computation. After frequent crashes of the restructuring tool, they decided to perform the change with `vi`. (They mentioned in the debriefing that the tool's style of automation encouraged them to make changes with the UNIX tools in a similar manner to the restructuring tool. Observations confirm this claim.) They used `vi` to search for references to `*line-storage*`, at which time they also discovered the `ls` alias. When they decided on a change to make, they ended up using regular expression matching to match (grouping uses task) and change all the similar expressions at one time (creating functions task). In one case, they used regular expression matches to find the candidate code, but then changed the code manually. After creating the functions, they found that one of their choices of abstraction, the `line-ref` function, was not used outside the module, and therefore was not strictly necessary. Their final interface was word-oriented.

The regular expressions they used for substitution were not trivial, and depended on the coding conventions of the program. Changing all expressions of the form `(list-ref *line-storage* lineno)` to call a new `line-ref` function was accomplished by a global substitution on only the first half of the expression, replacing "`(list-ref *line-storage*`" with "`(line-ref`". The transformation component of finding non-literal uses was also handled with global substitution by exploiting their observation that any parameter containing `*line-storage*` was named `ls`. Additionally, when planning the substitution of `ls` with `*line-storage*`, they determined that a similar variable name would falsely match `ls`, and so they temporarily "renamed" the similar variable with a global substitution before performing the main substitution. When they went to remove `*line-storage*` from the parameter lists, they only removed the variable from calls, and incorrectly left `*line-storage*` as the name of a parameter to each function.

Once D and M created the new module, they tested that the behavior of the code had not changed. After fixing syntactic errors, they modified the module to support the new data structure, then tested the new code to see that the enhancement behaved correctly.

4.2.4. Behavior of A and T (Text-based Restructuring Tool Team 2)

A and T first ran the program, then read through the code from top to bottom. During their first pass, they saw that some uses of `*line-storage*` occurred in function calls and were passed into functions. They recognized the non-literal uses and the mapping of `ls` to `*line-storage*`, and identified these uses by eye during their first pass through the code.

They then decided that they needed to look for each use of `*line-storage*` and replaced each use with a function. They read through the code, writing down likely abstractions as they saw them. Their abstractions were object-oriented, with `*line-storage*` or the `ls` alias being passed in at all points, removing the need to expose the non-literal uses with transformations. They identified implementations of abstractions such as get-word and length-of-line, but decided these would encapsulate too much if extracted as actual operations. This decision led them to a line-oriented encapsulation that does not include a length-of-line operation. After the second pass through the code, they began restructuring. Using the restructuring tool, they replaced instances of each abstraction with a new function and function call. They first tried to replace all uses of `(list-ref *line-storage*` *linenum*`)` with a call to an `ls-get-line` function, but they wanted to parameterize both `*line-storage*` and `line`, and as a result matched all `list-ref` calls, even those not representing the `ls-get-line` abstraction. They backed off this transformation, then started converting `ls-add-line`. Next, they created a single use of `ls-get-line` and used the Make Subcall transformation to convert computations identical to the function's body into calls on the function. Because they were performing the changes one at a time, they had to search through the text by eye looking for the next match. This worked, although when they finished transforming the uses, T mentioned that he thought another use existed. They searched for a moment before finding that no other uses existed. They then created an `ls-num-lines` function, and then changed `allwords` to be `ls-num-words`. Their "test" for completion at the end was the pronouncement, "we've created the four functions."

Once they finished with the change, they saved the file out and ran it to verify it still behaved as before. Next, they modified the source code to add the functionality, and finally tested the changes.

4.2.5. Behavior of J and K (Star Diagram Team 1)

J and K began by searching the code using the star diagram, navigating to references of `*line-storage*` in the code (finding variables task) while understanding the code. They moved linearly down the nodes on the first level of the star diagram, and also examined information on the right hand side of the star diagram to identify containing functions. In a later pass over the first level of nodes, they begin creating a function for each node.

They did not appear to look deeply into the star diagram. The choice of functions might suggest that the star diagram did not make clear to them that they could have chosen a word-oriented abstraction, although while looking at the text, they identified the more complex expressions as representing manipulations of words. They also decided early-on to create a line-oriented abstraction, so the choice of functions could be due to this, rather than limiting themselves to the first level of children.

At first, J and K implicitly took the star diagram's presentation of similar expressions (grouping uses task) as complete, and thus delayed undertaking the finding non-literal uses task with the needed transformations. When the non-literal uses were discovered in parts of the star diagram that they had passed over earlier, they exposed them with Inline Parameter. Because the star diagram did not have a direct way to turn the newly exposed expressions into calls on an existing function, they backtracked by inlining the function, thus putting all similar expressions in the star diagram together where they could be re-extracted. They expressed surprise over the fact that the star diagram did not show non-literal uses directly.

Despite the instructions, J and K created additional modules unrelated to the enhancement, and did not perform the enhancement. They did talk through how they would make the change, however.

4.2.6. Behavior of I and R (Star Diagram Team 2)

I and R made four understanding passes through the source code using `emacs` and the restructuring tool's text view. On the third pass, they identified that they not only needed to encapsulate the literal uses of `*line-storage*`, but also the implicit non-literal uses where `*line-storage*` was passed as an argument as well. After the fourth pass, they created a star diagram. They inspected all the direct uses of the variable in the star diagram, navigating to each use in turn. At each use, they decided how the code would need to change to support the new functionality they would be adding.

As they examined the uses of `*line-storage*` with the star diagram, they again noted the non-literal uses, and decided they could restructure the program to expose those uses. They used the Inline Parameter transformation to make the uses visible within the function `csline<=`. The star diagram exposed the multiple uses of `*line-storage*` within the function, but the star diagram doubled in size. Although they appeared to realize that the source code had been changed as they intended, they were disturbed by the increase in the size of the star diagram. If the star diagram increased in size, they presumed that they were moving away from their goal of "removing" all items from the star diagram. They backed out of the transformation using the undo feature, but then eventually reapplied the transformation.

They identified functions from the star diagram representation by first choosing the expressions that represented functions they understood, then dealing with the expressions they were not sure how to restructure. At one point, they used the interface list to determine that one needed function had already been created, but another had not. The last use they needed to encapsulate was an expression that represents the get-line abstraction, retrieving a specific word on a specific line. They seemed to identify the basic behavior of the code, but did not appear to understand that the computation has some sub-computations that must be parameterized, and stopped before they performed this transformation. They then talked through the code modifications needed to support the maintenance change.

5. Analysis

Our videotapes reveal that performing the global changes during restructuring required the programmers to make changes systematically, maintaining a constant awareness of their progress at multiple levels in the restructuring task. With each tool set, the programmers perform these bookkeeping activities differently in order to implement a change correctly. The correctness of a change depends on many factors, but two facets seen directly in the data are completeness and consistency.

By completeness, we mean that all items in a set are visited or changed. When understanding how a global data structure is used, the programmers must ensure that they visit and understand each use. When converting similar expressions into calls, the programmers must make sure that all similar sites are found and changed. When performing the overall restructuring action, the programmer must make sure that all uses of the data structure occur in one of the encapsulated functions. Achieving completeness is trivial if the programmer has an explicit list of uses and a method for visiting each.

By consistency, we mean that related but separate changes are applied in the same manner at different locations. For converting a frequently occurring expression into function calls on a new function, the programmer must ensure that the parts of each expression corresponding to the arguments on the new call must be correctly identified, and the argument order and types of each call site match the function definition's parameters. The programmer must also maintain consistency of the full interface during the change, ensuring that functions take arguments of the same type, have similar names, or correspond to an overall design. When a change is broken into temporarily separated operations, the programmer may forget a given constraint and create mismatches.

Completeness and consistency of a change depend primarily on maintaining information about the restructuring: parameter ordering, location of uses, functions to create, and common expressions that will be converted to function calls. Depending on how the programmers order the tasks they perform, they change the information they either must memorize, write down, embed in the tool, or lazily determine in order to perform a consistent and complete change. If they forget certain facts, cannot calculate them, or ignore them, then they can make mistakes, performing an incomplete or inconsistent change. Each tool set "remembers" different facts for the programmers, and the process the programmers follow maintains additional state and design information. We can see how the processes and tools retain certain information. When the tools and programmers' processes do not save information, we can see errors.

Completeness and consistency issues are complicated by the fact that it is sometimes costly to enumerate the set of related elements because some of the elements can be hard to find. This invisibility can lead to exploratory restructuring by the programmers. We also observed regular evaluation of progress by teams who worked in an exploratory manner. Such evaluations can lead to a change of direction or even backtracking.

We will examine first how programmers perform a single complete and consistent restructuring change. We then examine the programmers' exploratory behavior and evaluation of progress in the overall restructuring task.

5.1. Maintaining Completeness and Consistency During Modifications

To a first approximation, we observed two distinct completeness and consistency tactics among the six teams. One is based on visiting a set of potentially related items using the structure of the file; the other is based on visiting a set of related items using the underlying structure of their relationship.

5.1.1. The File-Based For-Each-Use Tactic

Both UNIX tools teams performed their work using the tactic of making linear passes through the file, inspecting or manipulating each use of `*line-storage*`in order to understand, group, and change the uses. By making modifications from top to bottom in the file, the programmers implicitly use their current position in the file to keep track of what uses have already been inspected, what uses are under consideration, and what uses remain to be considered. The position is represented by the current screen of displayed text, perhaps augmented by a cursor on the screen. This tactic is distinguished by the fact that the specific action performed at each use during a pass can vary depending on the kind of abstraction represented by the code surrounding the use. This tactic naturally guarantees completeness of the overall task (e.g., converting every use to a call), but it separates the handling of uses that are treated similarly (e.g., all uses that should be replaced by a call to the `get-word` function), complicating the consistency of changes related to a single abstraction.

These teams also broke down the creating functions task (creating a function and then making calls on that function) into subtasks that were handled by separate, complete passes (creating all the functions in one pass, then creating all the calls on those functions in a second pass). This subdivision helped the programmers group tasks with similar editing characteristics, but it distributed the changes related to a single function across multiple passes, complicating the consistent handling of the function from pass to pass. To help maintain consistency between passes, teams wrote down or memorized function names and parameter orders.

For example, F and B's use of `more` for searching dictated that they view the source in a largely linear fashion. They did not use `more`'s regular expression capability. Instead, they used `more`'s scrolling function and picked out uses by eye. By starting `more` on the program, inspecting each line on the current screen, advancing the screen, and repeating the inspection until the end of the file was reached, they guaranteed they would visit every use of `*line-storage*`. Although F and B wrote down the names of functions they created on a previous pass, they did not write down or correctly memorize parameter order between creating the functions and creating the calls. As a result F and B inverted the order of arguments when they created the calls. Such problems are indicative of the impact of separation due to linear visitation within a file or division of tasks into subtasks.

P and C at first used `emacs's` forward search to find each use of `*line-storage*`. Once they observed `*line-storage*`was passed to other functions, they searched for `ls`, the usual name of the parameter holding `*line-storage*`. They also wrote down the names of the functions containing the uses, perhaps to simplify searching for both `*line-storage*` and `ls`. They used the list of names to navigate to modification sites by searching for the function name, and visited functions in order of appearance in the file. Like F and B, P and C found all uses by visiting all lines, guaranteeing they found all locations where `*line-storage*` or `ls` were used. By creating the list of functions containing uses of `*line-storage*`, they simplified the searching process for subsequent subtasks.

The file-based for-each-use tactic can be summarized by the algorithm in Figure 7. Its

```
advance cursor to the top of the file
repeat
  search to next use of data structure or terminate
  determine the kind of use it is
  determine the action for that kind of use
  perform the action on the use
end
```

Figure 7. Algorithmic description of the file-based for-each-use tactic.

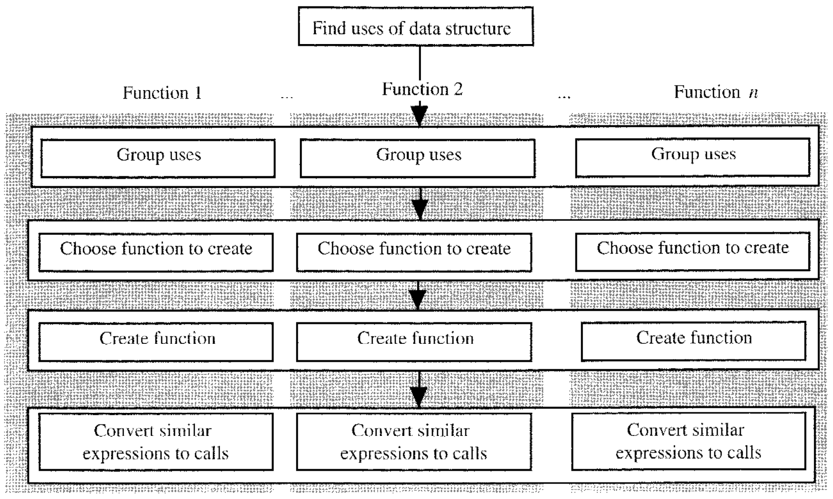

Find uses of data structure
Function 1
...
Function 2
...
Function *n*
Group uses
Group uses
Group uses
Choose function to create
Choose function to create
Choose function to create
Create function
Create function
Create function
Convert similar expressions to calls
Convert similar expressions to calls
Convert similar expressions to calls

Figure 8. Overall process of the UNIX tools teams, who used the file-based for-each-use tactic. Tasks are represented by boxes; arrows represent the order that they performed tasks. They created all functions in one set, then made a single pass through the code to convert the expressions into calls. The finding non-literal uses task is not shown due its widely distributed character.

influence on the overall process of the UNIX tools teams is shown in Figure 8. For a particular activity, all uses tend to be processed before another activity is undertaken.

Although only the text-based teams used linear passes through the file during the planning and execution of the transformations, all six teams resorted to the file-based tactic for ill-defined tasks such as figuring out the behavior of a component or developing an overall understanding of the system. More structured methods of visitation require a good understanding of what needs to be examined and what does not.

5.1.2. The Similarity-Based For-Each-Use Tactic

All four restructuring tool teams (including D and M, who ultimately used `vi` for the restructuring) used the tactic of inspecting or manipulating each conceptually similar use of `*line-storage*` in a single pass without concern for the dissimilar uses. Each team's use of its tool's pattern-matching and transformational capability ensured consistency of a single pass (e.g., replacing a group of similar uses with a call to `get-word`). Likewise, completeness is straightforward (e.g., after replacement, no uses match the similarity criterion). However, by performing all actions related to creating a new abstraction as a set, the programmers temporally separated design decisions about the functions, and thus sacrificed completeness and consistency of the overall change. Completeness of a task over all the uses (e.g., every use has been replaced by a call) requires a separate check to make sure that all the passes together touched on every use. (Of course, the star diagram's check is simple: the encapsulation is complete when the star diagram is empty.) To maintain consistency in the overall task of creating all the functions, the text restructuring tool teams sometimes wrote down details about function naming and parameter ordering to ensure the consistency of all the functions' interfaces. The star diagram teams, by contrast, recorded no such information.

D and M, who abandoned use of the restructuring tool part-way through their session, read through the file linearly for understanding, and may have grouped during this pass, but no information was explicitly recorded. They then used either the restructuring tool's "all expressions that match" feature of the Extract Function transformation or `vi`'s regular expression matching for the finding and grouping tasks. They also performed global substitutions to replace common expressions with calls to the new functions for all of the newly created functions but one. In this one case, the substitution was apparently too complicated to program, so they simply used the regular expression searching feature and performed the change by hand on each match, leaving consistency to the programmer. They also used structural properties of the program implementation to make correct global substitutions.

D and M tested for the completion of the encapsulation (the detecting completion task) by exploiting the linear structure of the file, searching for an unencapsulated use starting at the bottom of the new module (which is located at the top of the file):

M: Okay. So is . . . so are there any more references to line-storage?

. . .

D: So we don't want any above // lower than here [the functions encapsulating line-storage]. [D. searches for an occurrence of the line-storage variable.] And there aren't!

M: Okay.

D: That's good. So we've narrowed down the usage of that thing.

A and T, the second text-based restructuring tool team, first read through the file linearly to understand, then group, the uses. Unlike D and M, they completed the grouping uses task and wrote down a list of possible interface functions before beginning the restructuring. They used the restructuring tools to convert all expressions matching a code fragment into

```
while unhandled uses remain do
   choose a kind of use on the data structure
   determine the action for that kind of use
   for each use of the kind do
     perform the action on the use
   end
end
```

Figure 9. Algorithmic description of the similarity-based for-each-use tactic.

a function call before moving to the next function. In one case they did not use this feature because the tool matched too many expressions, and instead found each use by scrolling in the text interface and performing the transformations one-by-one.[6]

Star diagram team J and K used the stacking of similar expressions in the star diagram and the restructuring transformations to perform complete and consistent individual restructuring transformations, such as the creating functions task. The detecting completion task was achieved by noting that the star diagram for `*line-storage*` was empty. However, this team also used the graphical display for the grouping uses task and the choosing functions task. When browsing to find appropriate abstractions, they scanned linearly down the first level of children nodes in the star diagram, navigating to the program text associated wtih each node to look at the details. It appears, then, that they used the list of star diagram node stacks as a straightforward way of ensuring that they had visited every use, giving them a simple overall completeness check for these tasks even though they are using a similarity-based visitation tactic.

During the creating functions task we observed that although the programmers were aware that creating a function might miss some unexposed non-literal uses, and require the transformation to be redone later, they chose to create the new function anyway and move it into the interface list:

J: Well, should we do this `list-ref` while we're here? We can always undo it.

K: Yeah. That sounds good.

It appears that they are doing this transformation, in part, because of the ordering of the visitation. Their dialog suggests a notion of consistent movement down the star diagram: "while we're here." To skip over this item without transformation might be a violation of their overall task completeness tactic. On the other hand, by transforming only some of the uses, it appears they might be at risk of incompleteness of the single task of creating a function and all the calls on it. However, because the star diagram lists the created function in the interface view and would identify any unencapsulated uses related to this function in the star diagram view, this is not possible. Also, the star diagram's transformations and undo capability lower the cost of backtracking and making repairs.

Star diagram team I and R used the star diagram similarly for the purposes of completeness and consistency, but with two notable differences. First, to help them understand the

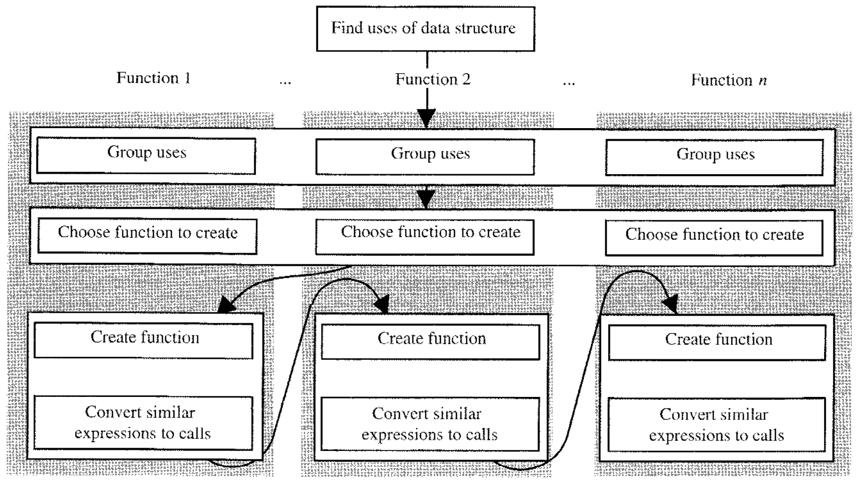

Find uses of data structure
Function 1
...
Function 2
...
Function n
Group uses
Group uses
Group uses
Choose function to create
Choose function to create
Choose function to create
Create function
Convert similar expressions to calls
Create function
Convert similar expressions to calls
Create function
Convert similar expressions to calls

Figure 10. Overall process of star diagram team 2, A and T, who used the similarity-based for-each-use tactic. The programmers reasoned about all the functions to create before beginning restructuring, but uses are replaced with calls on the appropriate function immediately after the function is created.

program, they explored the code with `emacs` and the text view of the restructuring tool before building a star diagram. Second, they actually examined and restructured nodes past the first level of the star diagram, signifying a slightly different enumeration tactic than J and K.

The similarity-based for-each-use tactic can be summarized by the algorithm in Figure 9. Its influence on the overall process of the restructuring tool teams is shown in Figure 10 and Figure 11. The process tends to group a function's creation with the creation of the calls on it, but separates the creation of the function.

5.1.3. Discussion

All teams, in different ways, employed ordered visitation tactics in order to keep track of what part of a task is done, what part is under consideration, and what part is yet undone. Depending on the specific tactic, completeness of an overall task or an individual subtask is ensured. Likewise, the order of visitation resulting from a tactic can aid consistency for a task by enumerating related elements together for simultaneous handling, or complicate consistency by not doing so. As one participant described a previous restructuring experience:

R: "If I change the order of parameters, I'll run through and make the changes in one pass, from beginning to end, and use the feature of emacs to find every instance of the thing. I sit there until it's done because, if I stopped, I'd be screwed. That's what I'd do to keep disaster from happening. That way, you can keep the change straight."

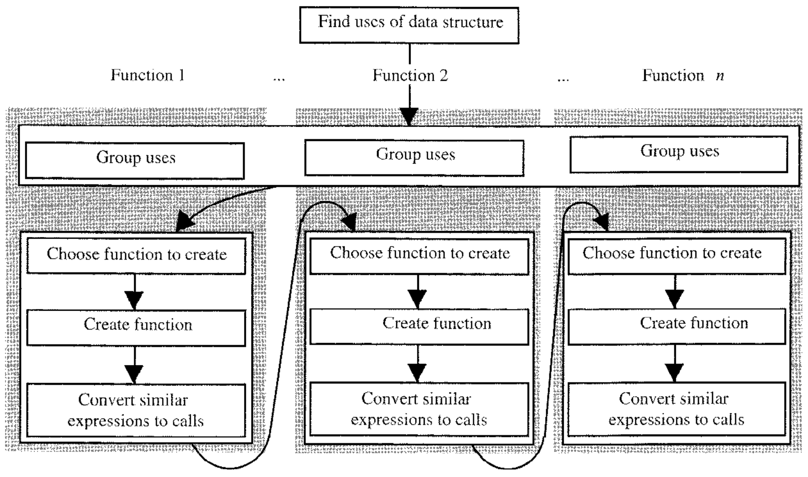

Find uses of data structure
Function 1
...
Function 2
...
Function n
Group uses
Group uses
Group uses
Choose function to create
Create function
Convert similar expressions to calls
Choose function to create
Create function
Convert similar expressions to calls
Choose function to create
Create function
Convert similar expressions to calls

Figure 11. Overall process of remaining restructuring tool teams, who used the similarity-based for-each-use tactic. The teams grouped similar uses before choosing functions (although some teams' delay in coping with non-literal uses qualifies this assertion), but then created each function in turn without consideration for the overall module interface. As a result, the teams sometimes made inappropriate decisions and had to back off of changes.

Although the tactic of enumerating all uses in each pass over the code does not appear economical, it is a simple tactic that is reasonably good at maintaining consistency and completeness of an overall task. In a program spanning multiple source files, this tactic can be accomplished with the UNIX tool `grep`, as long as the pattern for enumerating the uses is simple.

Visitation by stacking or regular expression matching of similar uses facilitated performing all tasks related to a given abstraction at once, maintaining consistency of issues related to that abstraction. In addition, the star diagram tool's enumeration of the node stacks themselves also aided completeness of the overall encapsulation overall task.

It is notable that the differences in behavior divided largely along the lines of UNIX tools teams and restructuring tool teams (both text and star diagram). However, it is also notable that one of the restructuring teams applied the similarity-based tactic using an editor, and that the restructuring tool teams resorted to the file-based tactic for ill-defined tasks. The restructuring tool teams also tended to have more problems with non-literal uses. Together, these suggest that the tool has some influence in programmers' behavior—sometimes positive and sometimes negative—but that the tools can be used in a variety of ways.

5.2. *Exploratory Versus Fully Planned Restructuring*

One aspect of programmer behavior not captured in the above analysis is that the overall process enacted by each team varied between a fully planned restructuring task (as shown in Figure 8 and Figure 10) and a rather exploratory approach to performing the restructuring (as shown in Figure 11). In this context, by exploratory we mean that activities such as examining all the uses, identifying the non-literal uses, choosing the functions, or stating an overall plan for the restructuring did not occur before actual restructuring began.

For instance, the teams who used the file-based for-each-use tactic planned out their restructurings before beginning changes. This behavior is strongly linked to their tactics. For example, replacing all the uses with calls on functions is not feasible unless all the functions have been planned.

A and T, who used the text-based restructuring tool, also took the approach of fully planning out their restructuring. Their behavior is surprising because the text-based restructuring tool does not require fully planning an encapsulation before beginning the transformations. It is trivial to create some functions before others have even been considered.

The remaining teams worked in a largely exploratory fashion. D and M, who used `vi` after abandoning the text-based restructuring tool, created one function after another without discussing an overall design or writing out an interface. However, this team was aware of the `ls` non-literal uses due to their early browsing, so knowing this information may represent a minimal kind of planning. I and R, the second star diagram team, also noted non-literal uses up-front during their early browsing of the code, but proceeded with their restructuring without recording or discussing a module interface or restructuring plan.

The most exploratory team was the first star diagram team, J and K. After one pass over the star diagram, they returned to the top of the star diagram and started choosing operations without any apparent plan. They had not spent enough time with the code in advance to discover the non-literal uses, and they had to backtrack due to their lack of planning on this matter.

These examples demonstrate, once again, that although the tactics used play a role in the overall process, the programmers still have the flexibility to choose a planned or exploratory approach—at least for a program of this size.

Originally, we thought we would prefer a process like that shown in Figure 10, which both chooses all functions beforehand and groups the creation of a function definition with the creation of its associated calls. However, in cases in which it is impossible or simply inconvenient to enumerate all the changes, exploratory design is attractive. For example, although the program restructured is small enough to plan out the encapsulation after exposing the non-literal uses, D and M perhaps felt fully planning the change would be more work than necessary.

In fact, the design of the star diagram (or any of the other tools) does not support a "planning before restructuring" model. The star diagram presents the program as it currently is, and provides no support for showing what it might look like in the future without actually changing the program. For example, transformations like Inline Parameter (used for removing non-literal uses), can dramatically reshape the star diagram as well as the code, but the star diagram prior to transformation gives little indication of how aspects like stacking will be affected. Even though the current design is explicitly presented, promising solution paths may thus be invisible to the programmer.

Although simple forms of non-literal uses can be exposed by building type-oriented as opposed to variable-oriented star diagrams (Bowdidge, 1995), the problem in general is much more complicated. There can be many ways of coding the same calculation; for instance, individual instances of the same calculation may be optimized on a case-by-case basis, obscuring their underlying similarity. Even though a star diagram would display such variants, their relationship might not be noted until late in the restructuring process if the star diagram is large.

Exploring a design in detail with these tools, then, is best aided by performing transformations to change the source code's form so that hidden issues will be exposed. Planning in this style is possible from the star diagram and text-based restructuring tools because they contain a multi-level undo mechanism (added after complaints in the pilot study) and transformations with inverses that allow retreating from a failed plan. All star diagram and text restructuring tool teams used this feature. Backtracking in the other tools is more awkward, but a few teams saved an intermediate version of their programs to allow backtracking.

In general, these problems point to an inherent unvisualizability of software (Brooks, 1987), and we believe that interleaved planning and restructuring may be more practical than complete planning, especially on larger systems.

5.3. Evaluating Progress

Teams that engaged in exploratory behavior also expended effort in evaluating how their work was progressing. Indeed, in an exploratory process, determining both that the work performed so far is adequate and what to do next is non-trivial and often not well-defined because the programmers are operating with incomplete information. Norman's model for planning and performing actions describes this as "perceiving system state" and "evaluating current state" with respect to the goal (Norman, 1986). An evaluation may result in noting a failure in prior planning, and backtracking may be undertaken to attempt another solution. As with the other issues discussed in this section, there are many ways that programmers can evaluate progress, and the tools being employed affect the evaluation process.

For example, D and M, using `vi`, had no easy way to evaluate a planned interface as they created the functions. As a result, when they began planning the enhancement to the restructured program, they found that their design for the new module contained a function that is actually unnecessary:

M: Do we ever need line-ref? like can it

D: Can we get away without line-ref? That's a good question. Let's find out.

D: I think the sort routine uses it, but// Whoa, only these guys [functions already in the line-storage module] see it. Ah, so that's good. So everything else it sees is only word-ref. So we don't need to worry about that hopefully here.

D: Okay, so um. . . But things do use line-length

M: Well we still have to write [reimplement for the enhancement] these two.

In contrast, A and T, who planned out their restructuring, used a list of functions as their to-do list, and assumed that when they reached the end of their list, they had completed the task. Because their modified program ran correctly, they had confirmation that they had not missed uses.

The star diagram teams, on the other hand, appeared to regularly evaluate their progress using the tool, observing the resulting code after every transformation and choosing the next appropriate transformation. For example, during most transformations, J and K clicked on a star diagram node, examined the text view to ensure they understood what the star diagram node represented, performed a transformation, then examined the source code to identify what the restructuring tool had done. These actions generally confirmed their beliefs. Their actions also strongly suggest that star diagram users need access to the source code for understanding the effect of a transformation and evaluating their progress towards a correct restructuring.

When programmers had no obvious cues for measuring progress, they sometimes chose inapproprite metrics. When I and R, the second star diagram team, performed the Inline Parameter transformation to localize uses of `*line-storage*`, the star diagram's size increased (because the transformation exposed many uses of `*line-storage*` in the called function). They assumed that they should be making the star diagram smaller in order to be progressing, and wondered if they were on the correct path.

They decide to undo the transformation, even though they realize that the code appears as they expect:

R: [The star diagram just increased significantly in size after inline-parameter] . . . but look what happened to the tree here.

I: Yeah, that looks like a bad thing to do.

R: Yeah, really. // hm that's really awful.

I: What did we do just now?

R: Well I think what we did was we took all the places that used to have csline and now made it be

[they apparently understand how the source code was changed, and that it did what they intended]

I: and put it // put the word // put the { textually function }

R: yeah, this is really funky, because all these parameters are functions, right and

I: Okay, let's undo that.

R: Yeah, definitely undo that.

I: Okay, that looked too // that looks bad // whatever we did {}

R: Yeah, definitely.

These examples show that programmers look for cues to assess their progress, and that the cues programmers use may be different than those intended by the tool's designer. Modifying misleading cues or training programmers to avoid using them might help.

5.4. Summary

These observations suggest that the tools and processes that programmers use affect the completeness and consistency of a modification. Techniques such as moving sequentially through the file or performing modifications in a logical order ensure completeness of changes during a single pass through the file, but may result in inconsistent changes applied during multiple passes through the file. Performing actions related to a single concept as a set can guarantee consistency of a specific modification, but require more effort to ensure that all modifications are performed. Explicit tool support, such as maintaining a list of change locations, can avoid some of these problems. However, as the complexity of the restructuring task grows, the programmer may not be able or willing to identify every modification site before beginning modifications, nor may the programmer be able to identify how each modification site must change. In such cases, the programmer may have to follow an exploratory process to perform a maintenance change.

The use of process to minimize consistency and completeness errors is reminiscent of Guindon's theory of opportunistic design (Guindon, 1990a). Guindon suggests that a designer sometimes may order actions during design to handle related concepts sequentially and minimize the details that must be remembered throughout the design process. As with programmers performing changes by abstraction, a designer who chooses to skip opportunistically into a different design task may risk losing track of the state of the design process and forget to return to the original task (Guindon et al., 1987).

Similarly, the restructuring tool teams' exploratory approaches to restructuring mirrors Schon's theory of reflection in action (Schon, 1982). Reflection in action asserts that professionals in many disciplines cannot solve problems with cookbook answers. Instead, they must explore the design space of the problem by creating a partial solution, evaluating whether the partial solution leads towards an appropriate solution, and refining the solution. The expert is forced to behave in this manner because the design space is so large that a complete solution cannot be immediately identified.

Both issues suggest that the problems encountered in restructuring may be endemic to any program modification that exposes new design issues during the change. Software tools for helping programmers maintain large programs may benefit from affordances for recording design state and supporting alternate task orderings for applying a change. Because all design possibilities may not be obvious before undertaking a change, tools should also permit exploratory approaches to maintenance and restructuring.

6. Generalizing the Results to Other Settings

From our observations of C programmers and from personal experience, we believe that the behavior of programmers restructuring by hand and programmers using restructuring tools matches how programmers work when restructuring larger programs. We expect programmers modifying larger programs would find ways to minimize the amount of source code they need to examine with tools such as `grep`, and would use printouts of source code and `grep` output to create markable "to-do" lists. For example, programmers examining a twelve-file C program used `grep` as their primary tool both to search the program during understanding phases and to find modification sites in the program. We observed cases where the programmers used a search pattern to enumerate all uses of a variable, then proceeded linearly through the `grep` output to guarantee visiting and modifying all uses of a given variable. Such behavior on the `grep` output is similar to how the UNIX tools teams in the Scheme study moved linearly through the entire file to guarantee completeness of a change. Similarly, a programmer making changes to the UNIX kernel told us how he used `grep` to find uses of variables and key field names, then used a printout of that output to visit each use.

Because `grep` sorts uses by location, not similarity, performing all changes related to a single abstraction (as the restructuring tool and star diagram teams did) would be more difficult. Personal experiences with printouts of source code show that programmers could still visit uses out of order and guarantee completeness as long as an external mechanism exists for marking each completed use and identifying when all have been changed.

The programmer may not be able to generate a `grep` search pattern to identify every site in the program that must be modified, due to either an incomplete design or lack of knowledge of the program. We expect that as systems become larger, tendencies towards exploratory behavior and evaluation will only increase. The increased distribution of information and the high cost of planning and maintaining information for restructuring will encourage, rather than discourage, such behavior.

7. Conclusion

Little is known about how programmers restructure programs, which hampers the design of usable tools for large-scale restructuring. We undertook an exploratory study by applying systematic observational techniques to six pairs of programmers in order to develop an understanding of the task of encapsulating a global data structure. Programmers used either UNIX tools, a text-based restructuring tool, or a restructuring tool employing the star diagram.

This study began as an assessment of whether the star diagram automated the right activities and displayed the correct information about the program. Although the answers to these questions seem to be positive, two other issues proved more important to investigate.

First, we found that the bookkeeping of the task's state, as supported by a tool's features, were influential to the programmers' behaviors. Specifically, the programmers followed processes and exploited features of the tools that allowed them to keep their place in low-level activities and in the overall restructuring. In essence, the tools not only provide a representation of the program, but also—in cooperation with the programmers—provide a representation of the task being performed.

Second, we found that the invisibility of design information played a large role in influencing programmers' behaviors. In particular, because the tools could not automatically display all the relevant information due to its disguised character, programmers chose either to take an exploratory approach to formulating the design or to make several passes over the code in order to assure themselves that they had found all relevant pieces of information.

Several discoveries led us to identify these issues. First, we had originally anticipated seeing five restructuring tasks being performed by the programmers, but during the coding of the transcripts we identified a sixth task, the finding non-literal uses task. Second, we had assumed an overall restructuring process that included a comprehensive pre-planning phase, but often found programmers restructuring in an exploratory fashion. Third, we found programmers performing activities in unexpected orders, and noted that these orderings simplified maintaining state during the modification.

Upon close examination of these orderings, however, we found a tension between the programmers' needs for achieving completeness and consistency of various kinds. Although each team's process ensured either all variable uses or all abstractions were visited, each process also separated the handling of related program elements, increasing the change of an inconsistent or incomplete modification. A process that guaranteed visiting every use of the variable (and thus ensured completeness of the overall task) separated actions occurring on the same variable use, risking inconsistent changes. A process that visited all uses corresponding to a given abstraction guaranteed the uses were modified in a consistent manner, but complicated testing that all uses of the variable had been modified.

The tension between completeness and consistency can be ameliorated if the programmers or tools correctly memorize or record where changes must occur and the form of these changes. For example, an editor's searching facility or the star diagram's stacks of similar expression support both completeness and consistency of individual changes by associating related program elements. The star diagram, by showing only unencapsulated uses of a variable, records the completeness of a given interface. However, none of the tools seem to provide comprehensive mechanisms for aiding the consistency of the overall interface for the new module.

As a consequence of this study and our predictions of how programmers might maintain large programs, we see three issues that a star diagram tool—-and perhaps maintenance tools in general—should address: maintenance of task state during modifications, exploratory approaches to restructuring, and invisibility of program details relevant to restructuring.

The importance of task information during maintenance suggests that future tools should maintain more information about the state of the modification in order to minimize the information the programmer must record or remember. Tools should also permit programmers to modify or view the source code in logical, not sequential, orderings so that programmers can focus on interdependent concerns at the same time.

Our observations of exploratory approaches to restructuring suggests the star diagram should support planning without the delays incurred to restructure the source code. We have proposed and implemented changes to the star diagram's features to better support the bookkeeping of design information in a star diagram tool for manipulating large C programs (Griswold et al., 1996). This tool helps identify abstractions and plan an encapsulation by allowing the programmer to annotate expressions as likely functions and remove them from the star diagram, but does not actually transform the source code. Becuase the tool does not need to perform the potentially expensive analysis for transformations, it can be used to examine large C programs. It also provides a primitive feature for building a star diagram of all variables of a given type, improving visibility compared to variable-oriented star diagrams. Newer versions of the tool have taken these ideas further.

The problems programmers faced with invisibility in the star diagram could be solved with training. For example, I and R identified a useful rule during their session. They recognized if they had followed a process of performing Inline Parameter until all uses had been exposed, then creating functions, they would have reduced the number of "invisible" uses. By providing such guidelines, novice users of the tool can recognize that not all uses of a data structure are visible from the star diagram, that the tool will not automatically handle such hidden cases, and that such uses should be exposed before restructuring begins.

This programmer study has raised many questions. The study was largely exploratory, and we substantially revised our initial assumptions and hypotheses. Moreover, the study was carried out with prototype tools, on a small program, in a laboratory setting, and on a small number of subjects. Further studies ameliorating these compromises will be required to test our new hypotheses.

Acknowledgments

Thanks to Christine Halverson for her advice and suggestions throughout the project. Thanks also to Edwin Hutchins of UCSD's Cognitive Science Department for guidance during this study. Nick Flor helped us plan and execute the C pilot studies, and Annavictoria Duyongco helped transcribe the C study videotapes. Our departmental staff, especially Jan Cox, Joanna Mancusi, Steve Hopper, and Dave Wargo provided valuable assistance. We are grateful to Larry Votta and Gail Murphy for their suggestions on the presentation of the experimental design and Peri Tarr for helpful comments. We also thank the anonymous reviewers who encouraged us to focus the paper's results. Finally, we're especially grateful to the anonymous subjects, without whom the study could not have been done.

Appendix

A. Subset of Task Instructions

You've been asked by your boss to make some modifications to KWIC, an index-creating program. The program is KWIC (Key Words In Context), found in the file kwic.s. KWIC takes as input a set of lines that make up the file to be indexed, and returns a sorted list of words in the file followed by the rest of the line. For example, if the input was:

```
the quick brown
fox jumped over the
lazy dog
```

the output would be the indexed word, followed by the remainder of the line the word appears on, followed by the prefix on the line.

```
brown the quick
dog lazy
fox jumped over the
```

```
jumped over the fox
lazy dog
over the fox jumped
quick brown the
the fox jumped over
the quick brown
```

A.1. Modifications to Perform

At initialization, the file read into an internal data structure: a list called `*line-storage*`. Each element of `*line-storage*`is a line in the input file, and each line is represented as a list of the words on that line. Your boss wants you to change the system so that the internal representation of the text file to be indexed is stored within KWIC as a list of words, with another data structure providing indexes to the beginning of lines. So, the above text file would look like:

```
'(the quick brown fox jumped over the lazy dog)
```

with the line data structure indicating that line 1 starts at word 1, line 2 starts at word 4, and line 3 starts at line 8. You're not allowed to change the format of the incoming text file.

This change is being made in anticipation of the next major modification to the system—filling lines of the file so that all lines are no more than a given size (such as 80 characters). You probably **won't** be asked to implement the line filling, but keep this next modification in mind!

You're also expected to test your code to make sure it works.

A.2. Preferred Method for Performing Modification

Your boss also has suggestions on how to make the changes. He's been hearing a lot about modularization and encapsulating data structures, and wants you to add the modification by first encapsulating the variables that are going to change—that is, hide the variable behind a set of functions BEFORE making the modification.

Creating a set of functions that hide accesses and manipulations of the variable improves the structure of the program. The restructuring localizes code related to the variable so programmers can easily find the code relevant to the variable. Changes are easier because all the code relevant to the variable is located in the module, instead of being scattered throughout the program. Thus, when you make a modification, you can consider the possible state of the variable only by the states the variable can put it in.

The process he wants can be described as follows:

- First, find the data structure or variables that need to change.
- Hide each data structure or variable behind a set of functions. This set of functions acts like a module interface. (Note that Scheme doesn't really have modules, but you can pretend the variable is only accessible from the interface functions, and nowhere else.)

- Make sure that your restructuring doesn’t change the running behavior of the code (i.e. you haven’t introduced any bugs.) At this point, you’ve got a program that behaves as it used to, but is better structured.
- Finally, add the planned enhancement and change the program’s behavior by modifying the functions encapsulating the program.

This approach will help you and future programmers understand and modify the code related to the variable.

B. Source Code for the KWIC Index Program

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;  INPUT MODULE
;;;
;;;  necessary if we’re not processing the revised scheme
;;; (define 1+ (lambda (x) (+ 1 x)))
;;; (define 1- (lambda (x) (- 1 x)))
;;;
;;;  list of list of words.
;;;
(define *line-storage* nil)

(define putfile (lambda (linelist)
  (letrec
    ;;
    ;; Adds a line. By convention, lineno=0 implies the first line.
    ;;  We assume the input is a list of symbols. We convert to strings
    ;;  for comparisons and such.
    ;;
    ((insline (lambda (line)
       (if line (set! *line-storage* (cons line (line-storage*))))))

  (do ((restlist linelist (cdr restlist)))
      ((null? restlist) nil)
    (insline (car restlist))))))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;  CIRCULAR SHIFTER
;;;
;;;   This module creates the illusion (or reality) the the
;;;   lineholder has had all the circular shifts of lines inserted for
;;;   all lines. For line i < j, all of i’s shifts come before the j’s.
;;;   The shifts are inserted in order starting from the original line.
;;;
```

```
;;;   Amazing fact:  for a line with N words, there are N circular
;;;   shifts.  This means that shift M is the line containing the Mth
;;;   word in the file, with the first word of the shift being the Mth
;;;   word.
;;;


(define *circ-index* nil)

;;;
;;;  Build an index of circulars.  These are represented as pairs of
;;;   (lineno, wordno).
;;;
(define cssetup
    (lambda  ()
      (letrec
       ((allwords (lambda (ls)
                    (do ((restls ls {cdr restls))
                         (sum 0 sum))
                        ((null? restls) sum)
                      (set! sum (+ sum (length (car restls))))))))
        (let ((numcslines (allwords *line-storage*))
              (cslineno 0)
              (numlines (length *line-storage*))
              (numwords nil))
          (set! *circ-index* (make-vector numcslines))
          (do ((lineno 0 (1+ lineno)))
              ((= lineno numlines) nil)
            (set! numwords (length (list-ref *line-storage* lineno)))
            (do ((wordsno 0 (1+ wordno)))
                ((= wordno numwords) nil)
              (vector-set! *circ-index* cslineno (list lineno wordno))
              (set! cslineno (1+ cslineno))))))))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;  ALPHABETIZING MODULE
;;;
;;;   This contains function alph.  It creates an array just like CS’s
;;;   but is sorted.
;;;

(define *alph-index* nil)

(define alph (lambda ()
  (letrec
    ;; Says if shiftno1 is less than or equal to shiftno2
```

```
   ((csline<= (lambda (shift1 shift2 ls)
     (letrec
       ;; Return the word on line number shiftno, at word number
       ;; wordno in the line.  Result is a string.
      ((csword (lambda (shift wordno ls)
         (let* ((lno  (car shift))
                (fwno  (cadr shift))) ; number of the first word
                                      ; in the shift
           (list-ref (list-ref ls lno)
                     (modulo (+ fwno wordno)
                     (length (list-ref ls lno)))))))

        ;; Returns the number of words in line number shiftno
        (cswords (lambda (shift ls)
                 (length (list-ref ls (car shift))))))

  (let ((lasti (min (cswords shift1 ls)
                    (cswords shift2 ls)))
        (result nil)
        (done? nil))
    (do ((i 0 (1+ i)))
        (done? result)
       (let ((maxed? (= i lasti))
             (cword1 (symbol->string (csword shift1 i ls)))
             (cword2 (symbol->string (csword shift2 i ls))))
         (if (or maxed? (not (string=? cword1 cword2)))
             (begin
              (set! done? t)
              (set! result
                    (if maxed?
                        (<= lasti (cswords shift 2 ls))
                        (string<=? cword1 cword2)))))))))))
(swap-indices (lambda (vec i j)
                (let ((temp (vector-ref vec i)))
                  (vector-set! vec i (vector-ref vec j))
                  (vector-set! vec j temp))))
;; Look at each cs-line from start to end and put its index in
;; the upper or lower half of *alph-index*. An equal comparison
;; defaults to the left side.
(qsplit (lambda (start end split)
  ;; start one below bot,and use bot as <= split
  (let ((low (1+ start))
        (high end))

   ;; swap the split and start so split doesn’t get mixed in swaps.
   (swap-indices *alph-index* start split)
   (set! split start)
   ;; do split
   (do ()
       ((> low high) nil)
     (if (csline<= (vector-ref *alph-index* low)
                   (vector-ref *alph-index* split)
                   *line-storage*)
         (set! low (1+ low))
         (begin
          (swap-indices *alph-index* low high)
          (set! high (1- high)))))

  ;; On exit of loop, we are guaranteed that (1- low) is in the low
  ;;  end. In the worst case, it is start (i.e., split).  So we swap
  ;;  this with the split, and then qalph will sort everything above
  ;;  and below (1- low).
  (swap-indices *alph-index* split (1- low))
     (1- low))))

    ;; Quicksort the shifted lines from start to end.
    (qalph (lambda (start end)
      (if (< start end)
          (let* ((split start)
                 (middle (qsplit start end split)))
            (begin
             (qalph start (1- middle))
             (qalph (1+ middle) end)))))))

   ;; THE REAL CODE
   (let ((numitems (length *circ-index*)))
     (set! *alph-index* (make-vector numitems))
     (do ((i 0 (1+ i)))
         ((= i numitems) nil)
       (vector-set! *alph-index* i (vector-ref *circ-index* i)))
     (qalph 0 (1- numitems))))))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; OUTPUT MODULE
;;;


(define allalphcslines (lambda ()
 (letrec
   ;; builds a circularly shifted line storage and a shiftspec
  ((csline (lambda (shift ls)
     (let* ((lno    (car shift))
          (fwno   (cadr shift))
          (wrdcnt (length (list-ref ls lno)))
          (revcs nil))
     (do ((i 0 (1+ i)))
         ((= i wrdcnt))
       (set! revcs
             (cons (list-ref (list-ref ls lno)
                   (modulo (+ i fwno) wrdcnt))
                   revcs)))
      (reverse revcs)))))

 (let ((numcslines (length *alph-index*)))
   (do ((i 0 (1+ i)))
       ((= 1 numcslines) nil)
     (write (csline (vector-ref *alph-index* i) *line-storage*)))))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; MASTER CONTROL
;;;

(putfile (list ’(a b c d) ’(one) ’(hey this is different) ’(a b c d)))
(cssetup)
(alph)
(allalphcslines)
```

C. Sample Transcript

This excerpt records how one of the star diagram teams performed two restructuring operations: Inline Parameter and Extract Function. The programmers first inline one of the parameters in the function definition for allwords, moving `*line-storage*`so that it is used directly in `allwords`, not passed in as a parameter. The second half of the transcript involves creating the `lines-in-file` function. First, the programmers identify that the length node in the star diagram maps to the expression (`length *line-storage*`) and represents the `lines-in-file` abstract operation. They then perform Extract Function. In this transcript, // represents a pause by the speaker, { } surrounds garbled or unintelligible words, **bold** indicates words spoken by the other subject at the same time, and [number] identifies where in the dialogue the numbered action occurs.

time	action	spkr	dialogue	comments
		J.	No, I think we had to go to the definition of allwords.	
	click on definition of allwords now visible in the star diagram	K.	Oh. . . What did I do? So // where's allwords?	K. says "Oh" before the transformation is complete and the star diagram has redrawn completely. Clicks on allwords to see the effect of the transformation.
7:21:30	look at code in text view	J.	{ } There we go.	
		K.	And so it did it.	
		J.	And that's what we wanted to do.	
	select allwords again	K.	Okay. So, now allwords is in our // um // we'll want to move that into the interface. Is that right?	
	press move into interface	J.	I believe so!	
7:21:50	[1] press length node in star diagram, [2] press extract function transformation button	K.	Okay. So we've got allwords and addline. That makes sense. Length [1]—piece of cake. We do the same thing. So we should maybe // maybe we should extract the function, [2] So uh line	
		J.	line-length?	
		K.	Uh. Okay.	
		J.	length of line?	
	press cancel in parameters dialog to close the dialogue prompting for information on the function to create.	K.	length—isn't it // is length the number of lines, or is it the number of words in a line?	Sounds like K. isn't sure what the node represents. Two lines down, there's the "Oh, length of line-storage" comment that makes me think he didn't really understand what the node mapped to.

time	action	spkr	dialogue	comments
7:22:20		J.	I think it's a	
	press extract transformation button again	K.	Oh, length of line-storage. So this is actually going to be	It isn't until he actually looks at the code in the text view—NOT in the parameter choosing window, that he actually seems to figure this out.
		J.	the number of lines	
	Type "number of lines" into name of new fn field. Press extract.	K.	the number of lines, so let's say numbef of lines. // Um, lines // extract.	
7:22:45	Transformation completes somewhere in here.	J.	So what did we do here when I wasn't looking?	
	click on numlines function call in star diagram, point at code	K.	Um so we made this numlines which is defined to be length of line-storage.	
		J.	Okay.	
		K.	And then	
		J.	What did we press to do that, I just // just so I know what we're doing?	
	press move into interface button	K.	I think we did **extract function**. extract function and so now I guess so now we move this into the interface.	

Notes

1. *affordance*: "The perceived and actual properties of a thing, primarily those fundamental properties that determine just how the thing could possibly be used." (Norman, 1989)
2. Although Scheme has a number of functional language features, it supports assignment to variables and pointer structures.
3. We have chosen the term *team* because the programmers are working together towards a common goal. Our use of the term should not be construed to mean that they normally work together.
4. For excerpts of transcripts in this paper, // indicates a pause by the speaker, {} indicates garbled dialogue or an uncertain transcription, and [] indicates programmer actions or editorial comments.

5. Although adding the new functionality before restructuring violates their directions when interpreted literally, it still meets the primary requirement that the enhancement and the restructuring are separated.
6. The restructuring tool supports the narrower match that they wanted, but not in a straight-forward manner.

References

Belady, L. A., and Lehman, M. M. 1985. Programming system dynamics or the metadynamics of systems in maintenance and growth. Research Report RC3546, IBM, 1971. Reprinted in M. M. Lehman, L. A. Belady, (eds.), *Program Evolution: Processes of Software Change*, Ch. 5, APIC Studies in Data Processing No. 27. London, Academic Press.

Blomberg, J., Giacomi, J., Mosher, A., and Swenton-Wall, P. 1993. Ethnographic field methods and their relation to design. In D. Schuler and A. Namioka, (eds.), *Participatory Design: Principles and Practices*, chapter 7, 123–155. Lawrence Erlbaum Associates, Hillsdale, New Jersey.

Boehm, B. W. 1975. The high cost of software. In E. Horowitz, (ed.), *Practical Strategies for Developing Large Software Systems*, 3–15. Addison-Wesley, Reading, MA.

Bowdidge, R. W., and Griswold, W. G. 1994. Automated support for encapsulating abstract data types. In *ACM SIGSOFT '94 Symposium on the Foundations of Software Engineering*, 97–110.

Bowdidge, R. W. 1995. Supporting the restructuring of data abstractions through manipulation of a program visualization. PhD dissertation, University of California, San Diego, Department of Computer Science & Engineering. Technical Report CS95-457.

Brooks, F. P. 1987. No silver bullet: Accidents and essence of software engineering. *IEEE Computer* 20(4): 10–19.

Chi, M. T. H. 1997. Quantifying qualitative analyses of verbal data: a practical guide. To appear in *Journal of Learning Sciences*.

Collofello, J. S., and Bortman, S. 1986. An analysis of the technical information necessary to perform effective software maintenance. In *5th Annual International Phoenix Conference on Computers and Communications*, 420–423.

Cousin, L., and Collofello, J. S. 1992. A task-based approach to improving the software maintenance process. In *Conference on Software Maintenance*, 118–126.

Curtis, B., Krasner, H., and Iscoe, N. 1988. A field study of the software design process for large systems. *Communications of the ACM* 31(11): 1268–1287.

Dumas, J., and Parsons, P. 1995. Discovering the ways programmers think about new programming environments. *Communications of the ACM* 38(6): 45–56.

Ericsson, K. A., and Simon, H. A. 1993. *Protocol Analysis: Verbal Reports as Data*, revised edition. MIT Press: Cambridge, MA.

Flor, N. V., and Hutchins, E. L. 1991. Analyzing distributed cognition in software teams: A case study of team programming during perfective software maintenance. In J. Koenemann-Belliveau, T. G. Moher, and S. P. Robertson, (eds.), *Empirical Studies of Programmers: Fourth Workshop*, 36–64. Ablex, Norwood, NJ.

Gray, W. D., and Anderson, J. R. 1987. Change episodes in coding: When and how do programmers change their code? In G. M. Olson, S. Sheppard, and E. Soloway, (eds.), *Empirical Studies of Programmers: Second Workshop*, 185–197. Ablex, Norwood, NJ.

Griswold, W. G., and Notkin, D. 1992. Computer-aided vs. manual program restructuring. *ACM SIGSOFT Software Engineering Notes* 17(1): 33–41.

Griswold, W. G., and Notkin, D. 1993. Automated assistance for program restructuring. *ACM Transactions on Software Engineering and Methodology* 2(3): 228–269.

Griswold, W. G. 1991. Program restructuring as an aid to software maintenance. PhD dissertation, University of Washington, Dept. of Computer Science & Engineering, Technical Report No. 91-08-04.

Griswold, W. G., Chen, M. I., Bowdidge, R. W., and Morgenthaler, J. D. 1996. Tool support for planning the restructuring of data abstractions in large systems. *ACM SIGSOFT '96 Symposium on the Foundations of Software Engineering*.

Guindon, R. 1990a. Designing the design process: Exploiting opportunistic thoughts. *Human-Computer Interaction* 5(2): 305–344.

Guindon, R. 1990b. Knowledge exploited by experts during software system design. *International Journal of Man-Machine Studies* 33(3): 279–304.

Guindon, R., Krasner, H., and Curtis, B. 1986. Breakdowns and processes during the early activities of software design by professionals. In *Empirical Studies of Programmers: First Workshop* 65–81.

Houde, S., and Sellman, R. 1994. In search of design principles for programming environments. *Conference on Human Factors in Computing Systems (CHI '94)*, 424–430.

Johnson, R. E., and Opdyke, W. F. 1993. Refactoring and Aggregation. In *Object Technologies for Advanced Software*, volume 742 of Lecture Notes in Computer Science, 264–278. First JSSST International Symposium.

Lange, B. M., and Moher, T. G. 1989. Some strategies of reuse in an object-oriented programming environment. In *Conference on Human Factors of Computing Systems (CHI '89)*, 69–73.

Letovsky, S., and Soloway, E. 1985. Strategies for documenting delocalized plans. In *Conference on Software Maintenance*, 144–151.

Letovsky, S. 1986. Cognitive processes in program comprehension. In *First Workshop on Empirical Studies of Programmers*, 58–79.

Lientz, B., and Swanson, E. 1980. *Software Maintenance Management: A Study of the Maintenance of Computer Application Software in 487 Data Processing Organizations*. Addison-Wesley: Reading, MA.

Miyake, N. 1986. Constructive interaction and the iterative process of understanding. *Cognitive Science* 10(2): 151–177.

Norman, D. A. 1986. Cognitive engineering. In D. A. Norman and S. W. Draper, (eds.), *User Centered System Design*. Lawrence Erlbaum Associates, Hillsdale, NJ.

Norman, D. A. 1989. *The Design of Everyday Things*. Doubleday: New York.

Opdyke, W. F., and Johnson, R. E. 1990. Refactoring: An aid in designing application frameworks and evolving object-oriented systems. In *Proceedings of the 1990 Symposium on Object-Oriented Programming Emphasizing Practical Applications*, 274–282.

Opdyke, W. F. 1992. Refactoring: A program restructuring aid in designing object-oriented applications frameworks. PhD dissertation, University of Illinois at Urbana-Champaign, Dept. of Computer Science, Technical Report No. 1759.

Parnas, D. L. 1972. On the criteria to be used in decomposing systems into modules. *Communications of the ACM* 15(12): 1053–1058.

Rosson, M. B., and Carroll, J. M. 1993. Active programming strategies in reuse, *ECOOP '93, 7th European Conference on Object-Oriented Programming*, 4–20.

Sanderson, P. M., and Fisher, C. 1994. Exploratory sequential data analysis—foundations. *Human-Computer Interaction* 9(3): 251–317.

Schneiderman, B., and Carroll, J. M. 1988. Ecological studies of professional programmers. *Communications of the ACM* 31(11): 1256–1258.

Schon, D. A. 1982. *The Reflective Practitioner: How Professionals Think in Action*. Basic Books: New York.

Strauss, A. L. 1987. *Qualitative Analysis for Social Scientists*. Cambridge University Press: Cambridge.

Sutcliffe, A. G., and Maiden, N. A. M. 1992. Analysing the novice analyst: cognitive models in software engineering. *International Journal of Man-Machine Studies* 36(5): 719–740.

Weick, K. E. 1968. Systematic observational methods. In G. Lindzey and E. Aronson, (eds.), *The Handbook of Social Psychology*, 357–451. Reading, MA: Addison-Wesley.

Wildman, D. 1995. Getting the most from paired-user testing. *ACM Interactions* 2(3): 21–27.

Robert Bowdidge is a Research Staff Member at the I.B.M. T.J. Watson Research Center in Hawthorne, N.Y. He received his Ph.D. in Computer Science in 1995 from the University of California, San Diego, and a B.A. in

Computer Science from the University of California, Berkeley in 1989.

Dr. Bowdidge's research interests center on improving how programmers maintain programs. His research has focused on how programmers plan changes and modify source code. He is also interested in restructuring tools, large scale program maintenance and improving commercial programming environments.

William Griswold is an Assistant Professor in the Department of Computer Science and Engineering at the University of California, San Diego. He received his Ph.D. in Computer Science from the University of Washington in 1991, and BA in Mathematics from the University of Arizona in 1985. He is a member of the program committee for the International Conference on Software Engineering in 1997 and 1998. His research interests include software evolution and design, compiler technology, and programming languages.