UNIVERSITY OF CALIFORNIA, SAN DIEGO

Supporting the Restructuring of Data Abstractions through Manipulation of a Program Visualization

A dissertation submitted in partial satisfaction of the requirements for the degree Doctor of Philosophy in Computer Science

by

Robert William Bowdidge

Committee in charge:

Professor William G. Griswold, Chair
Professor Edwin Hutchins
Professor Keith Marzullo
Professor Joseph Pasquale
Professor Richard N. Taylor

1995

The dissertation of Robert William Bowdidge is approved, and it is acceptable in quality and form for publication on microfilm:

Chair

University of California, San Diego

1995

For Christine

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

First, I must thank Bill Griswold for his advice and teaching over the last few years. He has taught me how to be a researcher, and has been a wonderful role model. All the members of my committee also made significant contributions to this work. Dick Taylor's questions about how the star diagram concept would scale encouraged me to create the C star diagram generator. Ed Hutchins inspired me to examine how programmers actually would use my creation. Joe Pasquale and Keith Marzullo reminded me of the real-world problems they faced.

Others have suggested productive lines of research. Jon Kay encouraged me to examine the UNIX kernel, and was always willing to discuss his personal restructuring experiences. Daniel Jackson inquired about the relationship between the star diagram and slicing, and encouraged further comparison of dataflow methods. My labmates, David Morgenthaler and Darren Atkinson, were extremely generous. Without their help and tools for parsing C and MUMPS source code, the scalability studies would not have been possible.

While every dissertation is too large a project to be performed in complete isolation, the programmer studies that form part of this thesis required more support from others than any project in my research career. The kind folk that provided assistance at one time or another ranged from departmental staff to the Human Subjects committee staff to colleagues who had conducted similar research. Most importantly, Ed Hutchins provided guidance as I learned about conducting such studies. Christine Halverson spent considerable time helping me learn about the field and plan the studies. Nick Flor helped me plan and execute the C pilot studies. Annavictoria Duyongco was in the unenviable position of being the first in our lab to "enjoy the experience" of transcribing videotape. Our departmental staff, especially Jan Cox, Joanna Mancusi, Steve Hopper, and Dave Wargo learned about new funding rules and pulled Ethernet cables into empty rooms to ensure a quiet location for videotaping.

Finally, I'm grateful to the anonymous subjects, who all gladly let me record their programming habits for posterity. Without their enthusiasm and professionalism, the study portion of this dissertation could not have been done.

This work was supported in part by NSF Grant CCR-9211002 and a SAIC Summer Fellowship.

VITA

November 22, 1966	Born, San Francisco, California
1987–1988	Project Engineer, Berkeley Softworks, Berkeley, California
1989	B.A., Computer Science, University of California, Berkeley
1989–1991	Research Assistant, Multimedia Laboratory, Department of Computer Science and Engineering, University of California, San Diego
1991	Teaching Assistant, Department of Computer Science and Engineering, University of California, San Diego
1991	M.S., Computer Science, University of California, San Diego
1991–1995	Research Assistant, Software Evolution Laboratory, Department of Computer Science and Engineering, University of California, San Diego
1995	Ph.D., Computer Science, University of California, San Diego

PUBLICATIONS

Robert W. Bowdidge and William G. Griswold, "How Software Tools Organize Programmer Behavior During the Task of Data Encapsulation", Technical Report CS95-443; Computer Science and Engineering Department, University of California, San Diego.

Robert W. Bowdidge and William G. Griswold, "Automated Support for Encapsulating Abstract Data Types", SIGSOFT '94: Symposium on Foundations of Software Engineering, New Orleans.

William G. Griswold and Robert W. Bowdidge, "Program Restructuring via Design-level Manipulation", Workshop on Software Specification and Design, Baltimore, Maryland, 1993.

P. Venkat Rangan, Walter A. Burkhard, Robert W. Bowdidge, Harrick M. Vin, John W. Lindwall, Kashun Chan, Ingvar A. Aaberg, Linda M. Yamamoto, and Ian G. Harris, "A Testbed for Managing Digital Video and Audio Storage", USENIX Summer 1991 Conference, Nashville, TN.

Thomas Schwarz, Robert W. Bowdidge, and Walter A. Burkhard, "Low Cost Comparison of File Copies", 10th International Conference on Distributed Computing Systems, Paris, 1990.

FIELDS OF STUDY

Major Field: Computer Science

Studies in Software Engineering.
Professor William G. Griswold

Studies in Multimedia Systems.
Professors Walter Burkhard and P. Venkat Rangan.

ABSTRACT OF THE DISSERTATION

Supporting the Restructuring of Data Abstractions
through Manipulation of a Program Visualization

by

Robert William Bowdidge
Doctor of Philosophy in Computer Science
University of California, San Diego, 1995
Professor William G. Griswold, Chair

With a meaning-preserving restructuring tool, a software engineer can change a program's structure to ease future modifications. However, deciding how to restructure the program requires a global understanding of the program's structure, which cannot easily be derived from viewing the source code.

We describe a manipulable program visualization—the *star diagram*—that supports the restructuring task of encapsulating a global data structure. The star diagram graphically displays information pertinent to encapsulation; direct manipulation of the diagram causes the underlying program to be restructured. The visualization compactly presents all statements in the program that use the given global data structure, helping the programmer to choose the functions that completely encapsulate it. Additionally, the visualization elides code unrelated to the data structure and to the task, and collapses similar expressions to allow the programmer to identify frequently occurring code fragments and manipulate them together. The visualization is mapped directly to the program text, so manipulation of the visualization also restructures the program.

We present the concept of the star diagram, and describe an implementation of the star diagram upon a meaning-preserving restructuring tool for Scheme. We also create star diagram generators for C programs, and test the scalability of the star diagram for large commercial C programs. Finally, we evaluate the star diagram's ability to assist data

encapsulation by observing programmers performing encapsulation using either standard UNIX tools or the star diagram.

Chapter I

Introduction

When a new tool or method is introduced to facilitate a specific task in the real world, that tool might not only provide a solution, but may expose unanticipated and previously hidden problems in the task being supported. We observed this effect when implementing a graphical user interface for a prototype restructuring tool. The restructuring tool allows a programmer to change the structure of a computer program without introducing errors into the program. Before implementing the restructuring tool, our concern had been to help the programmer perform the mechanical changes that the tool permitted. However, after implementing the the restructuring tool, we discovered that although the new interface helped us perform the changes, we discovered more difficult and fundamental problems: how do we decide what our goal for restructuring is, and how do we use the restructuring tool to accomplish this goal?

The research in this dissertation describes how a user interface for a restructuring tool can guide a programmer through changing the structure of a program. We will demonstrate an approach for guiding a programmer through the process of one kind of restructuring action: "encapsulating" a data structure. Specifically, the *star diagram* visualization supports encapsulation by providing a task-specific visualization of the program being manipulated with respect to the data structure being encapsulated. The star diagram also maintains state information needed by the programmer to identify progress in the task. Finally, by manipulating the visualization, the programmer causes the underlying program to be restructured.

We find data encapsulation to be a particularly beneficial restructuring task to support. Localizing the references to a data structure can simplify changes related to the design of that data structure. As a result, data encapsulation has the potential for being a driving paradigm for restructuring because many modifications can be considered as enhancements to existing data structures or variables [Parnas 79]. For example, Griswold and Notkin specifically used the concept of restructuring to achieve data encapsulation in a suggested process of adding an enhancement to an existing program [Griswold & Notkin 93]. According to this process, the programmer should first create a new data abstraction that localizes those design decisions formerly distributed across the entire program into a single module. Then, the programmer should modify the new module in order to add the enhancement. This process simplifies reasoning about, implementing, and testing the enhancement because the programmer only considers the behavior of the module in relative isolation, rather than trying to understand a large set of interactions between an unencapsulated data structure and the rest of the program. The concept of localizing the details in the program that are required to change [Parnas 72] can be a part of almost any maintenance task.

The star diagram is advantageous because it leads the programmer through this process by providing abstracted views of the code that highlight the details of the source code interesting to the programmer.

The contributions of this research are as follows:

- We show that the star diagram visualization can support the task of encapsulation.
- We show that the star diagram is useful for real programs written in common languages.
- We document how programmers restructure programs, and show that the star diagram changes the manner that programmers perform enhancement.

To introduce this thesis, we will first describe the motivation for the research, then describe the original restructuring tool that inspired the research problem. We will then describe why planning a specific restructuring is difficult with the tools that programmers normally use, then describe our solution. Finally, we will provide an outline of the thesis.

I.A Motivation for Restructuring

Maintenance of existing source code is responsible for a significant fraction of a computer program's lifetime cost. Any useful computer program is likely to require changes during its life in order to correct errors, support new peripherals, and adapt to changes in the domain served by the program. The exact cost of software maintenance varies with the kind of program, condition of documentation, and structure of source code, but the figures documented in the literature suggest the enormity of the cost. Lientz and Swanson observed that large companies spent 50% of their programming effort on maintenance [Lientz & Swanson 80]. Boehm documented an Air Force project where the cost to develop the program was $30 per line, and cost to maintain the program was $4,000 per line [Boehm 75].

Maintenance of software systems becomes more expensive over time as repeated modifications degrade the software's structure. Over time, programmers layer modifications upon the existing program to meet the needs and demands of users, resulting in more time needed to understand the code, apply the change, then fix bugs resulting from the change [Lehman & Belady 85].

Some of the structural problems are caused by modifications that do not fit well into the original design of the program. Although good software engineering practices encourage programmers to plan for future modifications, not every future design change can be predicted. Users may identify flaws in the program. Changes in the problem domain may require corresponding adjustments to the software. Even if the programmer knows also can be used to adjust a program's structure.

I.A.1 Restructuring by Hand

Programmers sometimes restructure existing programs by directly changing the code themselves in order to simplify a planned modification. However, restructuring by hand is both error prone and difficult because a programmer's changes can introduce subtle flaws and bugs into a previously working program [Griswold & Notkin 92]. As a result, programmers must work methodically and carefully to produce a correct, restructured program.

One example of restructuring by hand occurred during a fellow graduate student's research [Kay 95], where the structure of the networking code for the UNIX kernel proved inappropriate for an intended change.

The programmer was trying to change the format of the `mbuf` data structure in the kernel's TCP/IP networking code. (The code was derived from the 4.3 BSD UNIX implementation [Leffler et al. 88].) The `mbuf` data structures are blocks of memory that can be chained together in a linked list. These data structures permit efficient implementation of complex buffer operations, such as adding headers and trailers to the front and back of each outbound message. This ability to prepend and append bytes on a message allows the higher layers of the networking code to not worry about the size of headers and trailers required for a specific network device.

The existing implementation assumed that `mbufs` always point to buffer space in physical memory inside the kernel. The programmer wanted to convert the existing `mbuf` implementation to allow the `mbuf` data structures to also point to memory in the application program's memory (user space). This change would increase the performance of the networking code by eliminating the need for the kernel to copy the outgoing message between the application memory and the kernel. Because of efficiency concerns, the existing implementation of the `mbuf` data structure was not hidden from the rest of the system. Instead, the operating system directly manipulated fields of the `mbuf` data structure throughout the code. Some calculations were repeated in many places in the code to avoid procedure call overhead. Some parts of the operating system took advantage of implementation details, such as temporarily embedding information unrelated to `mbufs` inside an unused field of one `mbuf`. The functions allocating `mbufs` were accessed by multiple agents, operating at different priority/interrupt levels. As a result, any change to the `mbuf` data structure required ensuring that all parts of the networking code would correctly use the new `mbuf` data structure.

To add the new functionality to the networking code, the programmer created new functions for manipulating `mbufs` that pointed to memory in user space—manipulating the new kind of `mbufs` and mapping the pages of user memory into the kernel's memory space. The programmer then searched through the source code using the UNIX utility `grep` to identify code fragments that indicated locations where code might need to be changed, such as searching for the names of fields in the `mbuf` structure that would change. The programmer applied the correct modifications at each visited location.

The programmer required several days to perform the change, at all times being extremely careful that each modification would not introduce bugs into the modified code. After the modifications, the programmer had performed both the restructuring and the enchancement together, and could begin to debug the code.

This example points out that although restructuring by hand is difficult, programmers are willing to expend considerable effort to restructure a program if the restructuring will simplify a later modification. In this example, difficulty and scope of the global changes to the networking code could have introduced bugs into the code that could have been extremely difficult to trace. If the programmer had not fully understood the code before beginning to restructure, he could have missed the "tricks" used to increase performance or save code, and introduced additional latent bugs. However, the programmer was still willing to perform such a change. Although programmers are willing to restructure programs with only text editors, cross reference tools, program listings, and brightly colored highlighters, restructuring is by no means a simple task.

I.A.2 Tools for Restructuring

If the programmer wishes to restructure the program, but does not want to manually change the code, tools exist that can automatically and correctly adjust the structure of the program. Some software engineering tools automatically convert code into a supposedly more desirable form, such as changing goto-laden COBOL code into a structured programming style [Bush 85, de Balbine 75]. Although these tools may help programmers avoid the tedious aspects of maintenance and avoid introducing bugs into the source code, such tools shift decisions about the desired form of the program to the tool creator, not the tool user. These tools thus perform mechanical changes that satisfy an overall goal (such as "remove all gotos") without understanding the precise structural requirements needed for the planned change. For example, a program in which gotos only occur during exceptional events (such as jumping to the end of a procedure upon an error) might be more understandable than the same program in which loops and extra flags have been added in order to fulfill the goal of removing gotos. Better reasoning and understanding capabilities may help these tools understand when a given "bad" structure actually is worth preserving [Biggerstaff 89, Quilici 93, Soni 91]. Intelligent program analysis tools could recognize common cases where "poor" structure actually helps the programmer understand the code, and could then avoid restructuring such cases. However, because the decision of whether a given structure is "bad" or "good" relies more on how well the structure matches the user's conceptual understanding of the program, such tools would have to be extremely sophisticated to correctly decide when to restructure.

I.A.3 Meaning-preserving Restructuring Transformations

A third method to perform restructuring is to allow the programmer to guide automated tools. The maintainer can decide what sort of restructuring is needed for the specific change. An automated tool such as Griswold's meaning-preserving restructuring transformations can then perform the chosen change.

Griswold's restructuring transformations[1] allow the programmer to change the structure and code of a program in order to prepare the program for a maintenance change.

---

[1] Webster's 9th Collegiate Dictionary, *transformation:* "The operation of changing (as by rotation or mapping) one configuration or expression into another in accordance with a mathematical rule; esp: a change of variables or coordinates in which a function of new variables or coordinates is substituted for each original variable or coordinate."

The transformations are meaning-preserving; they change the structure of the program without affecting the program's output [Griswold & Notkin 93]. Ensuring the program's output remains constant guarantees the restructuring engine does not change the program's behavior in a manner detectable by an outside observer, and can thus be considered to appear unchanged.[2]

A prototype tool implementing these transformations allows a programmer to manipulate programs written in the imperative language Scheme. Scheme, a Lisp-like language, contains all the features of programming languages—pointers, side effects, and assignment—which make program analysis difficult [Dybvig 87]. The current restructuring tool supports twenty basic transformations [Griswold & Notkin 93], although composite transformations can be created from the existing transformations. The basic transformations can be divided into four categories.

- **Scoping transformations** change where functions and variables are defined and visible. For example, a function nested inside another (that Scheme, like Pascal, allows) can be moved out of the containing function with the move transformation.
- **Syntactic transformations** perform superficial changes to the program's parsed form. Renaming a variable or converting between kinds of local variable binding scopes (such as `let` scopes) are examples of such transformations.
- **Control flow transformations** affect the order in which statements occur. Examples include reordering statements in order to group related calculations together in the source code.
- **Abstraction transformations** create and destroy named objects in the code. Thus, the tool user can use the *extract-function* transformation to replace a common statement that occurs throughout the code with calls to a new function that performs the same computation. Figure I.1 provides an example of the extract-function transformation.

[2]The restructuring transformations, however, do not preserve the behavior of a real time system because they do not attempt to guarantee that execution time is maintained. However, one could in theory design a restructuring tool that maintained timing performance in addition to output behavior, assuming that a method exists for predicting timing constraints of a block of code.

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

(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 lnum))
...
```

(b) After extract-function

Figure I.1: The *extract-function* transformation is given a common expression occurring throughout the Scheme program, then 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.

Inversely, a function can be inlined, replacing calls to the function with a statement equivalent to the body of the function. The tool user can also replace a statement or expression with a new local variable initialized with the value of the original expression. Similarly, the programmer can replace uses of a function's parameter with the value passed to the function in its calls. Figure I.2 presents an example of the *inline-parameter* transformation.

These transformations often mimic common operations performed by programmers restructuring programs by hand.

The restructuring tool uses a program dependence graph (PDG) to explicitly represent control and data dependencies between operations of a program [Kuck et al. 81, Ferrante et al. 87]. The restructuring tool uses the dependence information in the PDG to check that a given transformation will not change the output behavior of the code being restructured. The restructuring transformations either check to see that changing the source code would not affect the program dependence graph's form (which guarantees

```
(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 I.2: The *inline-parameter* transformation takes a variable or expression that occurs as a parameter to a function call. It destroys the parameter in the function and arguments in all calls, and replaces uses of the parameter with the expression used as the argument in all calls. This transformation only works if all calls to the function have equivalent 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. Thus, the transformation brings the variable closer to the operations using it. (When programming in Scheme and Lisp, asterisks around a variable name are used as a convention for indicating global variables.)

---

the program's behavior will not change [Yang et al. 89]), or would change the program dependence graph in a manner that guarantees the behavior of the program would remain constant [Griswold 93].

I.A.4 Text-based Restructuring Tool

In order to simplify application of the transformations to maintain and restructure programs, we implemented a restructuring tool where the tool user selects and manipulates the source code of the program with a mouse-based graphical user interface. The restructuring tool lets the programmer (who, in our model, is also the tool user) view and manipulate the program through a window displaying the source code of the program (see the left side of Figure I.3). The tool user selects expressions in the program by clicking on the expression with a mouse. Because the transformations only manipulate syntactic units of the program, the source code view only allows such items in the program (such as tokens, expressions, and functions) to be selected and manipulated, much like some syntax-directed editors. Thus, the selection mechanism provides a forcing function to ensure the tool user

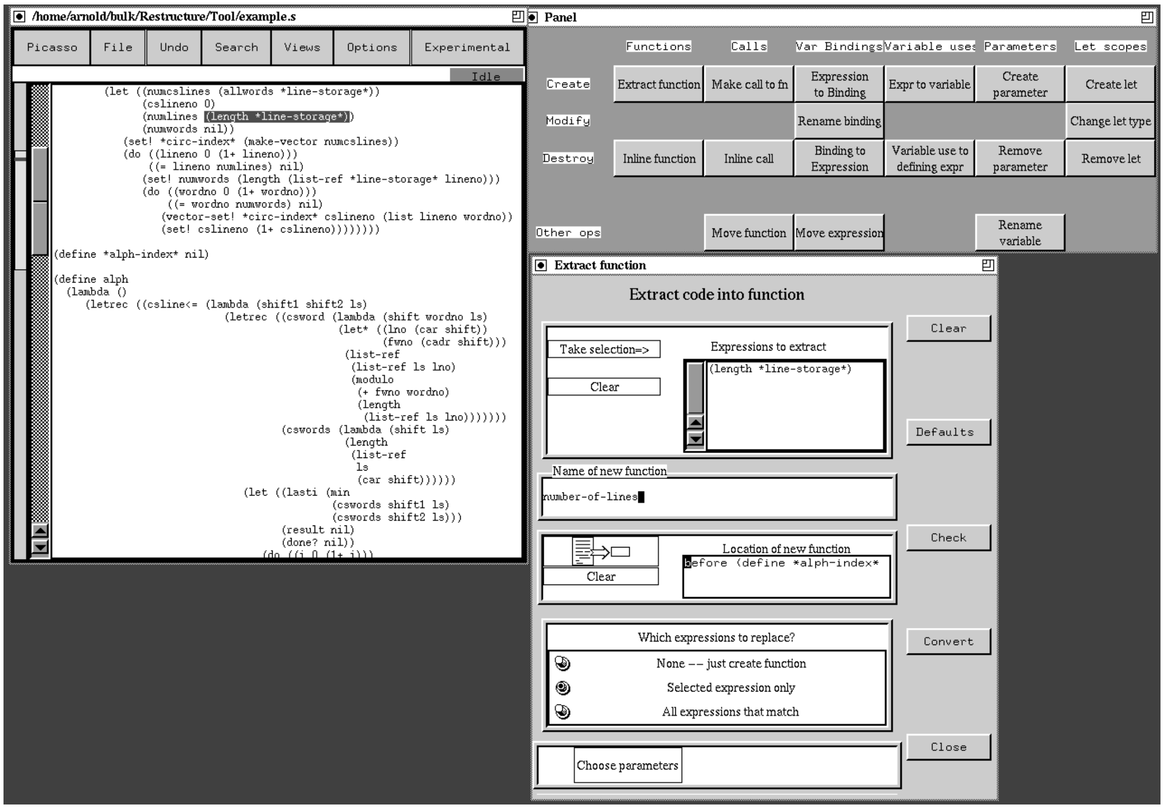


Figure I.3: User interface for the text-based restructuring tool that only presents the program in source code form. The left hand window contains the source code for the program being restructured. The window in the upper-right corner contains buttons representing each supported restructuring operation. The window in the lower right window is a transformation panel, where the user describes what the transformation should manipulate.

---

can never select an item that cannot be manipulated by the tool.

To perform a transformation, the tool user selects a transformation from a list of the transformations in the system (Figure I.4). The restructuring tool displays a separate dialog box for that transformation, and requests the user to fill in all the information needed to perform that transformation (Figure I.5). The tool user fills in the dialog box by selecting expressions in the source code or typing names directly into fields of the dialog box. For example, for the *rename-variable* transformation, the panel requests that the tool user select the variable being renamed and to type in the new name for the variable. The *extract-function* transformation panel, as shown in Figure I.5, requires more parameters. The tool user needs to select the expression to be converted into a function, type in the name for the function to be created, select the location in the source code where the new function will be placed, and describe the parameters for the new function. The user then presses the “Convert” button in order to start the transformation. The transformation first checks to make sure that the modification will not change the output behavior of the program. The transformation then changes the selected portion of the code as well as other related portions of the program in order to achieve the restructuring and guarantee output behavior is preserved. For example, to rename a variable, the tool user selects one occurrence of the variable, and the tool changes the name of that variable as well as all references to that variable. If the restructuring tool detects that the operation could change the output behavior of the code, (e.g. renaming a variable to a name already used in the scope) the tool prohibits the change and highlights the statements in the source code that create dependencies that prohibit the transformation. If the transformation succeeds, the tool user then examines the code to choose the next transformation to perform, and repeats the process until the program is in an appropriate form.

The restructuring tool is written in Allegro Common Lisp, and uses the PICASSO Application Framework to implement the graphical user interface portions of the code [Rowe et al. 91]. The tool runs on a Sun Sparcstation 10 workstation.

I.B Difficulties of Planning and Performing Restructuring

In our experience, we find the restructuring tool allows the programmer to concentrate on what the improved structure of the program should be and what restructuring transformations should be performed in order to get to that goal, rather than focusing attention on local modifications to ensure that bugs are not introduced into the code.

Planning restructuring, however, is not trivial. Consider the restructuring performed in the UNIX networking code example. The programmer may wish to replace some common, low-level manipulation of `mbuf`’s with functions such as “link these two `mbuf` structures”. To perform this task using the restructuring tool, the programmer must recognize the code that performs this operation, realize the code commonly occurs throughout

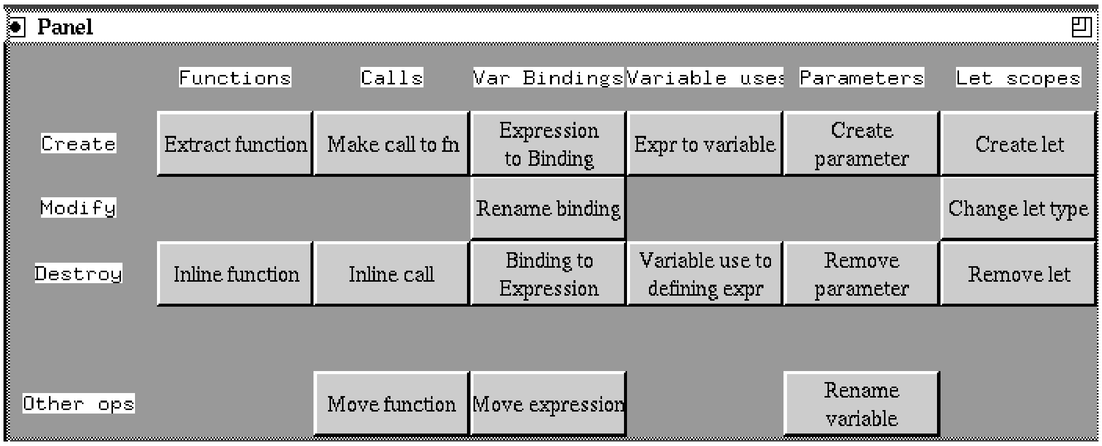


Figure I.4: Transformation panel lists the transformations that can be performed in the text-based restructuring tool interface. The columns of buttons group operations performed on the same kind of target expression; rows of buttons group similar kinds of operations.

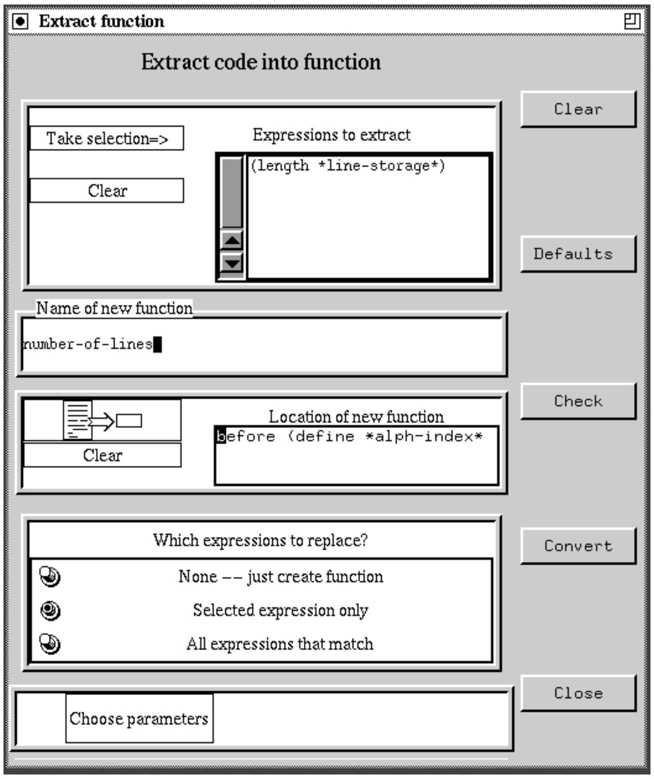


Figure I.5: Dialog box for performing the extract-function transformation.

the program, and should be replaced by function calls. The programmer must then search through the source code view identifying all occurrences of the common operation, either by eye or with some sort of searching utility, and decide on a form of the body for the new function and choice of parameters that captures the behavior of all occurrences of the expression.

Creating a function thus requires the programmer to gain a *global* understanding of the system in order to identify a common statement that should be hidden within a function, and to understand *in detail* exactly how the common statement appears in all its contexts in order to correctly propose a function that can replace all the occurrences. Although the source code view of the restructuring tool supports understanding the details of the program, the programmer will need to memorize what the relevant statements scattered through the code look like in order to maintain the global understanding of how the expression is used.

The significant modifications to the source code caused by each restructuring transformation also challenge the programmer's understanding of the global and local aspects of the program. For example, a simple transformation that inlines all calls to a specific function significantly changes the structure of the program. Even a few transformations of this form could completely invalidate the programmer's understanding of where a given data structure is referenced, or knowledge of the calling hierarchy of the program. As a result, the programmer might need to carefully examine the code after each restructuring to understand the current state of the program's structure, and check at each step for progress towards the final goal of a suitably restructured program. This need for the programmer to maintain both a global and detailed local view of the program in the presence of rapid change is a key obstacle for planning restructuring.

I.B.1 Sample Problem: Bus Route Simulator

In order to demonstrate the concepts in restructuring presented in this chapter, we will present an example of why a program might need to be restructured, and discuss how the changes would be made.

`Bussim` is a small Scheme program that simulates a bus traveling around a circular route, printing out the location of the bus after each mile traveled. (The code for `bussim` is provided in Appendix B.B.) The bus's location is encoded by the value of the `*route-distance*` variable[3]; it contains the distance in miles that the bus has traveled.

Although the existing program is useful for modeling how the bus travels on its route, the bus company employees using the software might also be concerned with how far the bus has traveled recently. For example, the users might want to simulate maintenance and repair activities when the bus has traveled a given number of miles. The programmer could decide to add a `*distance-traveled*` variable to capture this behavior. Performing this modification requires the programmer to examine all the source code and decide where the new code should be added. However, analyzing the code in this way would require understanding the entire program. Such an understanding task is difficult even for programs of moderate size.

I.B.2 Encapsulation: A Specific Restructuring Task

Alternatively, the programmer could encapsulate an existing and related data structure into an opaque structure called an abstract data type, then change the behavior of that abstract data type to include the new functionality. This approach follows the data abstraction style of designing programs [Parnas 72, Liskov et al. 77]. An ADT is a programming language structure where the details of the data structure's implementation are be hidden from the rest of the program. Only a small set of interface functions should be allowed to access the data structure directly; other parts of the system should call the interface functions to manipulate the data structure. Using an ADT simplifies changes to the data structure because the programmer needs to only understand how the interface functions manipulate the data structure, rather than identify every possible access of the variable throughout the code.

Creating a new abstract data type in a program requires choosing an appropriate

[3]By convention, global variables in Scheme programs are indicated by names beginning and ending with asterisks.

set of interface functions for that ADT. In general, the set of functions should be intuitive, orthogonal, minimal, and insensitive to anticipated changes [Lampson 84, Parnas 79].

The functions forming the operations on the abstract data type should also be *cohesive*. Cohesion and its complement, coupling, are subjective criteria that can be used when evaluating a system's structure [Stevens et al. 74]. Cohesion measures the degree of "functional relatedness" [Sommerville 89] between two components. To maintain high cohesion for an ADT, all functions and variables within the ADT should perform operations directly related to the abstract data type's purpose, and nothing else. For computations within each function to be cohesive, the function should perform a single operation on the ADT. Coupling is an indication of inappropriate connection between program components. To ensure low coupling for an ADT, the ADT should not refer to the internals of other modules, or perform actions related to the purpose of another ADT. Although quantitative techniques for measuring cohesion exist, we use the terms only to describe a programmer's subjective design goal.

For an example of encapsulation, we can consider a our enhancement task for `bussim`. By looking at its source code (Appendix B.B), the programmer may recognize that the variable `*route-distance*` already represents the bus moving mile by mile. The programmer could thus consider manipulations of `*route-distance*` to also imply manipulations of a new variable, `*distance-traveled*`, that counts the total mileage traveled by the bus. The programmer could simply examine the code, find every occurrence of `*route-distance*`, decide the context of the use, and decide how `*distance-traveled*` needs to be manipulated in the same context. For example, when `*route-distance*` is incremented (meaning the bus has moved a mile), `*distance-traveled*` probably also needs to be incremented. However, when `*route-distance*` is set to zero, the context could either be at initialization or to reset the trip counter when the bus returns to its starting point. Although this change appears simple for a small program, examining every use in a larger program would be difficult.

Instead, the programmer could encapsulate `*route-distance*`. Then, to add the `*distance-traveled*` variable, the programmer examines the module encapsu-

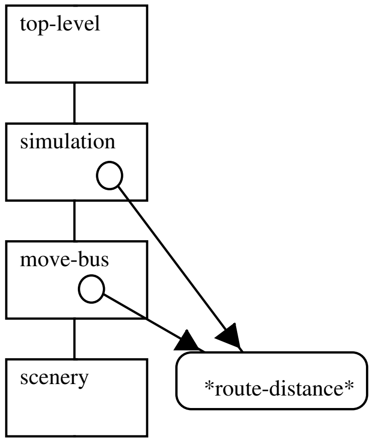


Figure I.6: Structured design view of a program where all functions directly access the global variable `*route-distance*`. In this diagram, Boxes represent functions, circles represent location of one or more uses of a given variable, and rounded boxes represent variables. Lines between boxes represent calls; variable uses are connected to the variable accessed. Note that because most functions in the program access `*route-distance*` directly, any change to the form of `*route-distance*` will require global changes to the program.

---

lating `*route-distance*`, and determines how `*distance-traveled*` relates to each operation. As long as the functions chosen to encapsulate `*route-distance*` encapsulate the semantics of the variable well enough to allow insertion of the `*distance-traveled*` variable, and a meaning-preserving tool exists to ensure the encapsulation does not introduce errors into the code, this modification will be simpler than changing the entire program because less code will need to be examined to add the enhancement.

Figure I.6 shows the bussim example in its initial form, where `*route-distance*` is used directly by functions in the system. The suggested structure appears in Figure I.7. In the improved structure, the code related to the `*route-distance*` variable is isolated from the rest of the code, and so the programmer can simply examine the interface functions to determine how to add the `*distance-traveled*` variable to the system.

Encapsulation can be useful not only in programs originally designed in a func-

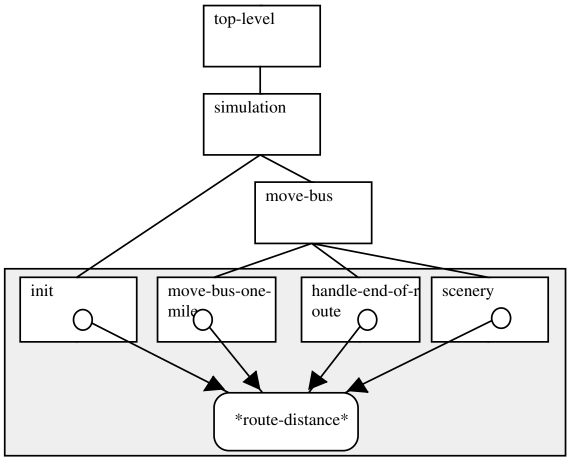


Figure I.7: Preferred implementation of the `route-distance` data structure. In this case, accesses to the variable only occur in the interface functions. As a result, adding functionality concerned with how far the bus has traveled only requires changing code within the module.

---

tional decomposition style (where variables are accessed by all parts of the system), but also for programs written in a data encapsulation or object-oriented style. Although such programs would appear to have all data structures already encapsulated, several situations can encourage restructuring such programs. An unencapsulated data structure could be added to the system as an afterthought, or could have started as a simple data structure, and then evolved into a more complex object. Two encapsulated data structures may actually represent a common abstraction that must be combined into a single module. A data abstraction could include inappropriate behavior, requiring changing the boundaries of the encapsulation. Finally, one abstraction might actually represent two separate abstractions. Any of these scenarios is possible during the design of modern computer programs, and motivates the need for encapsulation.

I.B.3 Difficulties of Encapsulating from the Source Code

Although performing the encapsulation of the `*route-distance*` variable may appear trivial from the high-level picture, planning an encapsulation from the source code proves more difficult in practice. The source code proves inappropriate because the programmer cannot easily find the references to the data structure and identify the common calculations that form likely interface functions.

For example, the programmer could find references to `*route-distance*` simply by reading through the source code. Programmers sometimes search code and plan modifications on printed source code listings when planning a modification because the technique displays the context of uses, supports odd searches that might not easily matched using regular expressions, and supports "exploring the code" without a specific focus of interest. However, the technique scales badly as programs become large because manually searching is time consuming and awkward. Overlooking instances of the code of interest is likely. Matching similar expressions on paper listings requires comparing expressions scattered across multiple pages, requiring the programmer to memorize and recall the context of variable references in order to group them into candidate interface functions.

Some programmers might use cross reference tools such as the UNIX utility `tags` to find all uses of a variable. However, `tags` provide little support for encapsulation because the tools only provide navigation to individual uses, and do not provide a global, overall view. Because the programmer needs to look at all references to identify similarities, keeping the occurrences of the data structure scattered only complicates the programmer's task.

More commonly, programmers use a text searching tool such as `grep` [Aho 80] to provide assistance in locating direct references (Figure I.8). To encapsulate a variable, the programmer searches for the variable name, and `grep` returns a global view of how that variable is used throughout that program, formatted as a list of relevant lines in the program. However, `grep` does not understand the underlying programming language, and so searches may match variables with similar names in other scopes, or miss indirect references to a data structure (such as when the variable is passed as a argument to a function, and the argument is manipulated within that function). `grep` also proves less useful for encapsulating types because no single search can identify all uses of all variables of a given type. `grep` also cannot easily highlight nearby, cohesive statements to provide

```
1:(define *route-distance* 0)
18:     (write *route-distance*)
19:     (scenery *route-distance*)
20:     (set! *route-distance* (1+ *route-distance*))
21:     (if (> *route-distance* 40)(set! *route-distance* 0))
27:     (set! *route-distance* 0)
```

Figure I.8: Using `grep` to search for all uses of the `*route-distance*` variable in the `bussim` program.

---

relevant context important to the programmer; instead, `grep` only shows a fixed number of nearby lines. Even if these weaknesses are resolved, simply listing all the computations that reference the data structure potentially involves a substantial amount of text. Such a solution does not show the similarities among calculations, which the programmer must still identify by memorization and recall. Finally, `grep` does not assist in the actual manipulation.

These methods all have the weakness that they provide no global view for helping the programmer match common expressions and identify likely interface operations throughout the program. When identifying which statements occur commonly in a large program, the programmer will have to recall what has already been seen, rather than having the information constantly visible. If some similarities are identified early-on, but then more are discovered later, a previously defined operation may need to be generalized to accommodate the new discovery. This in turn requires updating the calls to the generalized operation. Without additional assistance, a programmer must methodically search the source code to identify all computations on the data structure and keep careful records, or otherwise risk performing an ill-planned encapsulation.

The scattering of references throughout multiple files may instead encourage a programmer to build the interface based on recall of the common expressions, or make interface design decisions locally. Both situations could result in the programmer creating the necessary module and changing all the code, only to find that a necessary function was left out of the interface, or that a given function was unnecessary. The programmer may then end up with an interface where some functions are defined at a different “level of abstraction" than others. If restructuring is already in progress, the programmer might find it too time-consuming to correct these conflicting interface decisions, because the affected code is distributed throughout the system. As a result, the programmer may end up with a module interface that is not orthogonal, minimal, or complete. Because restructuring aims to ensure the program's structure and module interfaces fit a clean structure, creating interfaces without sufficient planning can simply substitute one substandard structure for another, without making the program any simpler to maintain.

I.C Solution: The Star Diagram

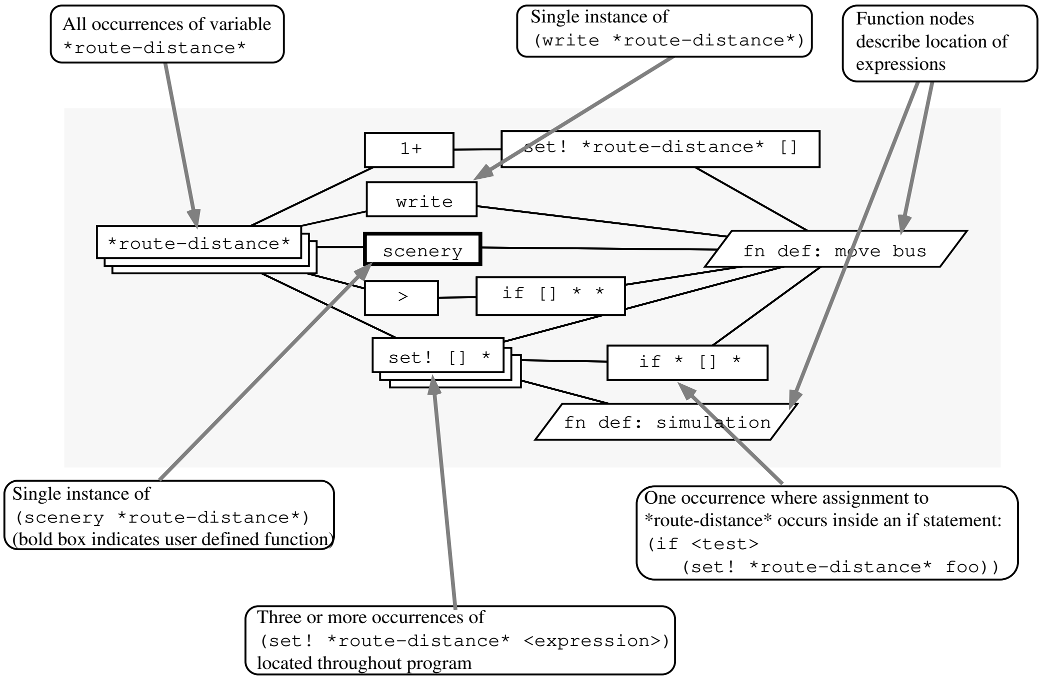
Our question, therefore, is how do we help programmers perform encapsulation with a restructuring tool? The problems with the tools programmers currently use suggest a set of requirements for an "encapsulation tool". The approach should provide a global, condensed view of the program. The programmer should be able to gain an understanding of the code from a single view, without having to search through the code. The view should elide details of the program irrelevant to encapsulation. The view should also help the programmer match and group similar expressions when identifying likely functions. The tool should have knowledge of the source code's language in order to avoid incorrectly identifying variable uses and provide some indication when cohesive calculations are split across multiple statements. Finally, we assume that a programmer performing modifications best understands what the final form of the program should be. Therefore, we want the system to provide the information that the programmer needs to plan the interface, not formulate the plan and perform the changes automatically.

Our solution, called the *star diagram* (see Figure I.9) follows all of our design guidelines for an encapsulation tool: present a global yet focused view of how a data structure is used in a program, group similar calculations, present details of the relevant code, allow restructuring, and let the user make the design decisions. The star diagram presents the programmer with a graphical representation of uses of the data structure, finding all uses of the data structure, then matching and grouping the similar expressions to identify likely functions for the new module's interface.

To demonstrate the basic star diagram concept, we will show how the star diagram can be used to plan the encapsulation of the `*route-distance*` variable from the `bussim` example. The star diagram identifies all possible expressions in the program that contain references to `*route-distance*` (Figure I.9). The star diagram presents the statements in a tree-like structure. The root of the star diagram represents all uses of `*route-distance*` throughout the program. Nodes connected to the root represent operations where references to `*route-distance*` occur. Edges in the graph link operations consuming the value of the previous calculation, and represent larger and larger expressions and statements using the data structure's value. Stacked nodes indicate multiple similar operations that occur throughout the code.

By examining the first set of nodes to the right of the `*route-distance*` node, we can see the operations using the variable directly. `*route-distance*` is incremented by the `1+` operation, printed out by the built-in `write` operation, passed as a parameter to a user defined function called `scenery`, and has its value tested by the greater-than operator. The stacking at the `set!` node suggests that the assignment is a common operation, and thus may be a candidate expression to turn into an interface function for the module encapsulating `*route-distance*`. Because there is no stacking on nodes to the right of the `set!` node, we can assume that the assignments occur in dissimilar contexts. Nodes to the right of the first "level" of nodes indicate the larger expressions that consume the result of the previous operation. For example, the `set!` node to the right of `1+` indicates that the `*route-distance*` variable is incremented exclusively inside of an assignment statement of the form `(set! *route-distance* (1+ *route-distance*))`. This expression would appear to represent the "move bus one step" function, and to check this hypothesis, we click on the `set!` node. The restructuring tool then highlights the relevant code in the source code view represented by the star diagram node.

With the knowledge from the star diagram, we can make a guess about a possible interface for `*route-distance*` by examining the frequently occurring operations,



Source code for `bussim`:

```
(define *route-distance* 0)
(define scenery (lambda (mileage)
  (cond ((= mileage 0) (write "Bus Depot"))
        ((< mileage 10) (write "Road to Big City"))
        ((< mileage 20) (write "In Big City"))
        ((< mileage 30) (write "Road to Village"))
        ((= mileage 30) (write "In Village"))
        ((< mileage 40) (write "Road to Small City"))
         (else (write "In Small City")))))
(define move-bus (lambda ()
  (write "Distance: ")
  (write *route-distance*)
  (scenery *route-distance*)
  (set! *route-distance* (1+ *route-distance*))
  (if (> *route-distance* 40) (set! *route-distance* 0))
  t))
(define simulation (lambda ()
  (let* ((i 0))
    (set! *route-distance* 0)
    (do ((i 0 (1+ i)))
        ((= i 100) nil)
      (move-bus)))))
(simulation)
```

Figure I.9: Star diagram for the `*route-distance*` variable in the `bussim` program. Brackets indicate that the node to the left occurs in the given place in the expression. Star represents a wild card, and indicates parameters of an operation that may vary.

and identifying the expressions performing important calculations on the variable. (See Figure I.10). We will need a function that writes the number of miles traveled along the route to replace the calls to `write`. We will need a function that prints the name of the bus's current location. The `scenery` call already performs this task, but in order to encapsulate `*route-distance*` into a module with a set of functions hiding it from the rest of the system, the uses of `*route-distance*` should occur directly within the `scenery` function, rather than passed into it. Because the > (greater than) node compares `*route-distance*` to "40" (the length of the bus route, as identified from comments or documentation), the test represents the end of route test. However, the enclosing expression, represented by the `if` operation, actually does the comparison, then sets `*route-distance*` to zero if the bus has reached the end of the route. So, we could choose instead to create a "handle end of route" function that combines both calculations. Next, we could choose the 1+ operation as the "next mile" function, or get the `set!` surrounding it, that corresponds to "move bus one mile". Of the three occurrences where `*route-distance*` is set, one occurs in the "move bus one mile" function, and another is within our "handle end of route" function. The remaining instance occurs at the beginning of the program, and represents the "initialize" concept. We would thus choose an interface with five functions: `scenery`, `write-mileage`, `move-bus-one-mile`, `handle-end-of-route`, and `initialize-route-distance`.

The programmer enhancing the `bussim` program could then use this restructured program to add the `*distance-traveled*` variable. The programmer can search through the new module enclosing `*route-distance*` and decide when `*distance-traveled*` should also change. The `*distance-traveled*` variable should be initialized during the `initialize-route-counter` function, and incremented whenever the bus moves, but should not change when the `*route-distance*` is reset at the end of a route. The programmer thus can change only the `move-bus-one-mile` and `initialize-route-distance` functions to add the enhancement.

The example shows the most important properties of the star diagram. The

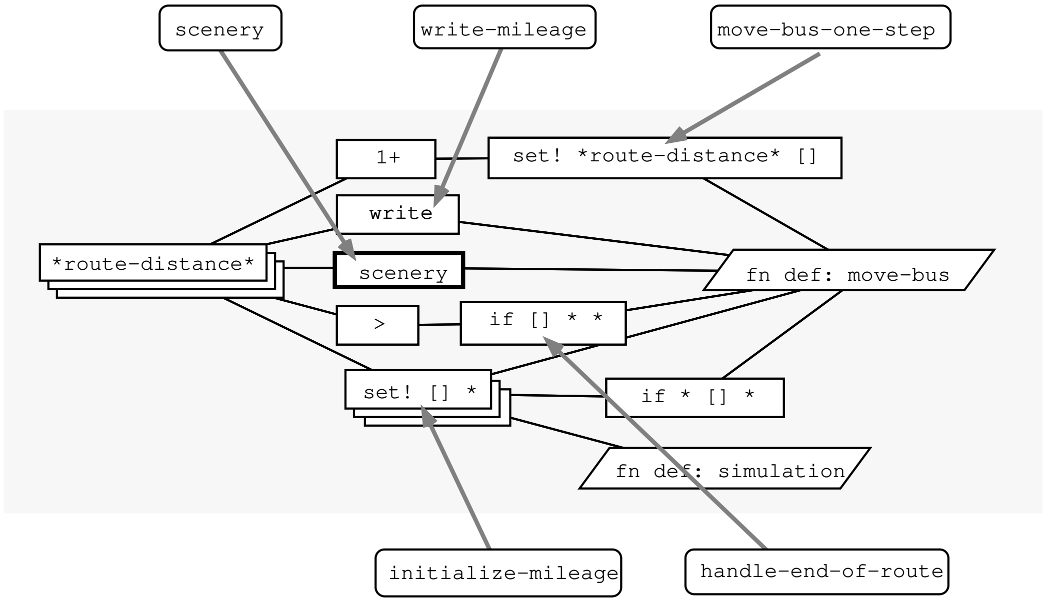


Figure I.10: Possible interface functions for an abstract data type encapsulating the `*route-distance* variable`. Arrows connect the name of new functions with the code instantiating the function. The bold line around the node for `scenery` indicates that `scenery` is a call to a user-defined function.

programmer can immediately perceive how the data structure is being used throughout the program, identify stacking of nodes and branching of the graph to identify likely interface functions for the new abstract data type, and select nodes in the star diagram view to examine the relevant source code.

In the actual star diagram, we would select nodes in the star diagram, then select restructuring transformations to actually change the source code. As restructuring transformations are performed, the star diagram updates the view so the programmer always understands the current state of the program. In a larger example in Section II.D, we will provide an example of how the programmer restructures through the star diagram view.

I.D Organization of the Dissertation

The thesis of this research is that the star diagram can help programmers perform encapsulation by presenting the details of the program relevant to encapsulation, and allowing the programmer to perform restructuring changes directly from the star diagram.

To study whether the claim is valid, we will examine the star diagram concept from three perspectives. First, we will argue that the star diagram presents information programmers would logically need to perform encapsulation, and that this information is difficult to identify from the source code view (Chapter II). We will present a framework for constructing star diagrams (Chapter III), then use the framework to describe how extensions to the star diagram concept can make the star diagram more useful (Chapter IV).

Second, we will argue that the star diagram will work in real environments (Chapter V). We will show that the concepts work for languages other than Scheme, first showing that we can create star diagrams for other languages, and that these star diagrams can be used to plan restructuring. Specifically, using the framework presented in Chapter III, we will implement star diagram generators (without the underlying restructuring tool) for the programming languages C and MUMPS. We will use these star diagram generators to examine medium-sized (hundred thousand lines) commercial and public domain programs (`emacs` and the SunOS networking code), and a half-million line MUMPS program. These case studies will allow us to highlight some of the difficulties we encountered when creating large star diagrams, and then argue why the star diagram concept should scale to larger programs.

Next, we will observe how programmers actually use the star diagram to perform restructuring, and find that the star diagram and restructuring tool significantly change how programmers perform restructuring. (Chapter VI).

We will finally describe how the star diagram relates to other concepts in the software engineering community (Chapter VII).

Chapter II

The Star Diagram Concept

In this chapter, we introduce the star diagram in detail. We describe the star diagram visualization in detail, explain what the star diagram displays, provide an example of its use, describe the motivation behind some of the design choices, and argue that the star diagram satisfies our design criteria.

Our implementation of the star diagram is built on top of the existing Scheme restructuring tool, and thus displays and manipulates programs written in Scheme. The Scheme star diagram is designed to help the programmer identify how to encapsulate a single variable representing a global data structure. Our prototype only encapsulates variables because the Scheme language does not support explicit types and pass-by-reference, and so we could not easily mimic encapsulating a data type. As we will see in Chapter IV, variants of the star diagram support encapsulating multiple variables and types.

II.A Motivation for Design

As described in Section I.C we expect that an encapsulation will require the following properties:

- **Global, condensed view of program:** Programmers cannot quickly examine all lines of the program, and so any condensed representation that can reduce the number of lines they have to examine will simplify the task. The programmer should also be able to gain an understanding of the code from a single view that identifies the similarities among pertinent elements, not by scrolling through the code. The reduced view should simplify planning the restructuring as well.

- **Elide details irrelevant to the encapsulation:** The program contains many facts irrelevant to the specific encapsulation. The tool user should not need to sort through these in order to plan the encapsulation.

- **Grouping similar expressions:** In order to identify candidate functions for the interface of the abstract data type (ADT), the programmer needs to identify similarities between expressions scattered throughout the code.

- **Support restructuring:** The view should permit restructuring the program using the restructuring tool.

- **Programmer control:** Because the tool user is restructuring a program to support understanding and maintenance, the tool user best understands what the final form of the program should be, and should be left in control of planning and manipulation.

We will demonstrate that these criteria are satisfied in a demonstration of the star diagram.

II.B Example: Parnas's KWIC Program

Before describing the star diagram and performing a sample encapsulation, we will propose a sample problem to motivate the planned encapsulation. We have a sample program, Parnas's KWIC indexing program [Parnas 72], which creates a permuted index of a text file. Figure II.1 shows a sample input and output for the KWIC indexing program. KWIC's source code is provided in Appendix B.A.

The starting implementation of KWIC is written using a functional decompositional strategy, In a functional decomposition, the programmer creates functions based on a stepwise refinement of the task to be performed, breaking a large problem into component

Input: text file	Output: permuted index
``the quick brown``` ```fox jumped over the``` ```lazy dog```	```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```

Figure II.1: Sample execution of the KWIC Index program.

---

tasks [Bergland 81] [Stevens et al. 74]. The entire program operates by performing four major tasks when creating the index. First, the program reads the input text file into memory, then "rotates" the lines to create the circularly-shifted lines. The program then sorts the lines and finally outputs the index.

One of the side effects of functional decomposition is that the different tasks communicate with each other via shared data structures. Parnas points out that the weaknesses of a functional decomposition is that any change to the data structure format requires global changes to the program [Parnas 72]. In this case, the three globally-visible arrays allow the tasks to share information. The arrays are called `*line-storage*`, `*circ-index*`, and `*alph-index*`. Figure II.2 shows a graphical depiction of the structure of the current KWIC implementation, where the global variables are directly accessible by all parts of the system.

One specific drawback of this implementation is that the text file is stored in an array, `*line-storage*`, with one line per array entry. Because the entire file has to be in memory, the size of text files to be indexed must be smaller than the computer's memory. We would prefer KWIC to handle arbitrarily large files, and must redesign the internal representation of the text file, perhaps changing the *line-storage* concept so that lines in the text file are always accessed from disk, or by providing a caching mechanism to keep

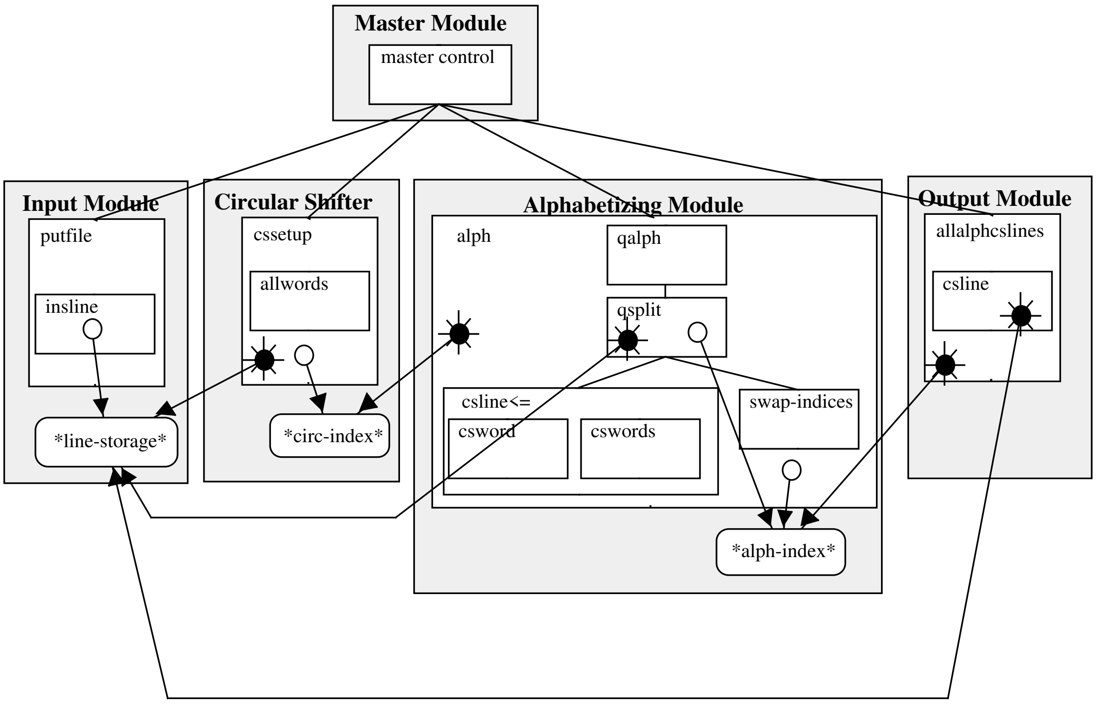


Figure II.2: Original structure of the KWIC Index program, using a notation similar to the Structured Design notation. The stars indicate variable accesses across module boundaries—in this case, the places where the arrays *line-storage, *circ-index*, and *alph-index* are being accessed directly.

---

some lines in memory.

If we wish to change the format of the `*line-storage*` array to support these modifications, then we have two possible plans. First, we could make the change by finding every occurrence of `*line-storage*` throughout the code, and identify the purpose of the access. We then decide how the code at each use needs to be changed to cache the information on disk. However, such an approach distributes the design decision about how the lines are stored throughout the code. Because we will be making global changes to the program, we have a greater chance of accidentally introducing bugs.

A better approach would be to encapsulate the `*line-storage*` data structure inside an abstract data type, with a set of interface functions hiding the internal representation from the rest of the program. Once we have encapsulated the variable, then we could simply recode the interface functions to add new functionality, and avoid disturbing the rest of the existing source code.

We will use the example of encapsulating `*line-storage*` both when explaining the appearance of the star diagram and demonstrating its use.

II.C Description of the Star Diagram

Before planning a restructuring task, we will describe the star diagram and how the tool user (assumed also to be the programmer performing the modifications) encapsulates variables using the star diagram. We will describe all facets of the star diagram in detail, first describing the star diagram visualization itself, then the features of the star diagram window, then the transformations.

II.C.1 Overall description of the Star Diagram

Figure II.3 shows an actual star diagram window, being used to encapsulate the `*line-storage*` variable in the KWIC program. The star diagram window is made up of four parts. The star diagram visualization itself fills most of the window, and appears as the directed graph in the bottom half of the window. A "button bar" above the star diagram visualization provides a button for each restructuring transformation that can be performed from the star diagram. To the left of the window, a scrolling list specifies the functions currently in the interface of the ADT being created. Finally, along the top of the window, a set of menus gives access to less frequently used commands.

To perform a restructuring action, the tool user selects a node (box or parallelogram) in the star diagram graph representing an expression or statement in the source code that the tool user wants to manipulate. The tool user then presses the button corresponding to the desired restructuring transformation. Dialog boxes prompt the tool user for additional parameters to the transformations, such as the name and parameters for a function to be created. Once the tool user has specified any additional parameters, the transformation immediately checks to ensure that the changes applied to the program will preserve meaning, and if so, then the transformation manipulates the source code of the

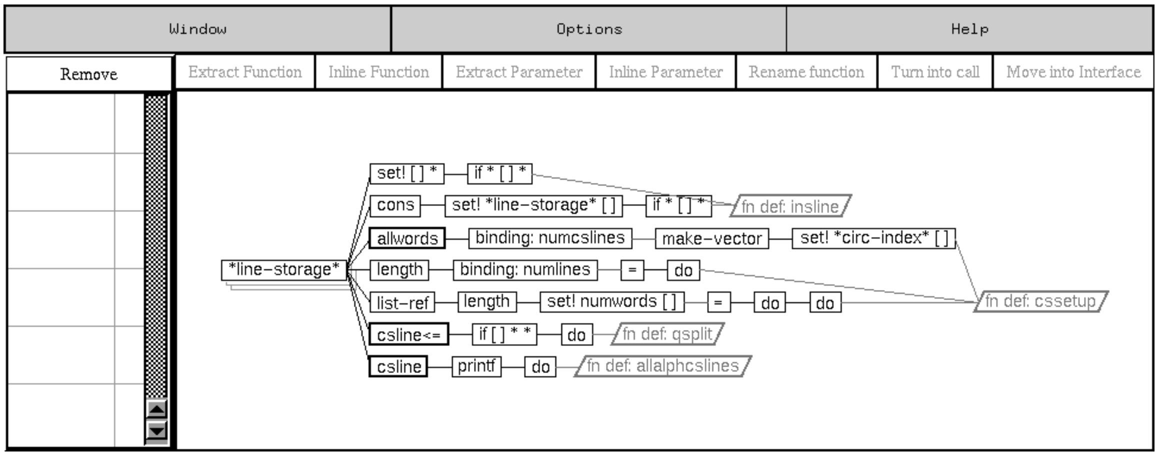


Figure II.3: Star diagram for the array `*line-storage*` in the KWIC program. In the left column, the scrolling list displays the current functions in the interface, and the Remove button at the top allows the tool user to remove a function from the interface so it can be manipulated using the star diagram. The main window consists of the seven transformation buttons at the top, and the star diagram in the rest of the window space.

---

program. The restructuring tool then updates the star diagram window to show the current state of the program. As the tool user exposes functions that should be members of the ADT's interface, the programmer uses the "Move Into Interface" transformation button to move the function's definition into the module interface, and places the name of the function in the list of interface functions. All uses of the variable being encapsulated that occur inside a function in the interface are removed from the star diagram because the uses are no longer unencapsulated. Thus, the star diagram explicitly maintains a list of what uses of the variable have been encapsulated, and which have not. The Remove button at the top of the scrolling list removes functions from the interface if the tool user finds that a function should not be in the interface.

II.C.2 Star Diagram Visualization

The star diagram visualization itself appears in the lower portion of the star diagram's window.

To a first approximation, the star diagram is like a data flow graph [Ferrante et al. 87], with nodes of the star diagram corresponding to operations,

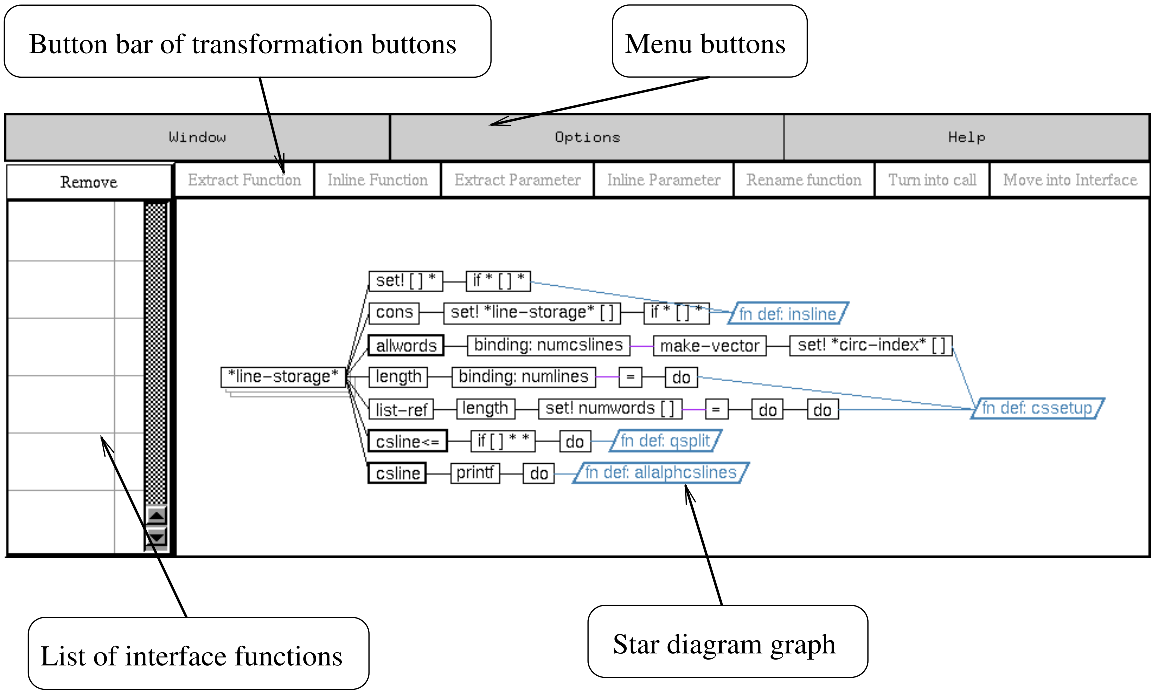


Figure II.4: Major parts of the star diagram window.

---

and edges in the star diagram corresponding to data flow between operations. The star diagram differs from a data flow graph by only showing the data flows generated by references to the data structure being encapsulated, rather than dataflow relations for all variables in the program. The star diagram also represents all uses of a single variable as a single node–the root—in the star diagram. As a result, the star diagram appears as a tree of computations, with branches diverging when variable uses occur in different computational contexts.

- Starting at the left, there is a single root node denoting all the references to the data structure itself.

- Operators directly referencing the data structure are represented by nodes connected to the root by edges denoting the reference. An operator is any of the language's basic constructs, such as addition, assignment, and function call.

- In turn, for each such node, the operator consuming its result is indicated by the node connected to it.

- When two nodes denoting the same operator are connected to the same node on the left (starting at the root), then they are stacked on top of each other. This stacking (represented as shaded boxes appearing behind nodes, as in Figure II.5) is what gives the star diagram its tree shape.

- The tree is terminated at the level of a function body.

The resulting tree of nodes in the star diagram thus contains all of the computations that refer, directly or indirectly, to the data structure. A path from any node to the root of the tree denotes a unique computation on the data structure. For instance, in a star diagram for `*line-storage*` (Figure II.3), the path

`*line-storage*` → `list-ref` → `length`

denotes the Scheme expression

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

which computes the number of words on the specified line of text[1] Similarly, each node in the tree possibly represents the implementation of an abstract operation on the data structure. Nodes near the left of the tree are expressions or statements that may implement "low level" operations encapsulating little more than the data structure representation itself. Nodes farther to the right could implement "higher level" operations that also encapsulate how some semantic properties are realized from the data structure.

Overlapping similar nodes has two key advantages. One, the size of the resulting tree is reduced by eliminating the redundancy, improving the tool user's global perspective on the encapsulation. Two, the overlapped nodes can be collectively extracted as a new function definition, saving the tool user the effort of searching for and manipulating the nodes individually. For instance, in Figure II.6, the

`*line-storage*` → `list-ref` → `length`

path is actually a path of stacked nodes whose corresponding expressions come from the functions `cssetup`, `csword`, and `csline`.

[1]We will usually use the arrow shorthand given above when referring to individual nodes to the star diagram. Because many nodes can share the same label, this approach allows us to unambiguously specify a node.

The function definition nodes serve two purposes. First, they provide context about where the uses of the variable occur. Second, the star diagram algorithm stops at function boundaries. Although we could have star diagrams where nodes might represent calculations split across function boundaries, we assume programmers think of calculations within a particular scope, and so representing and manipulating calculations that cross function boundaries could be confusing.

In order to provide additional context and improve the manipulation of computations, the star diagram differs from a data flow graph in two main regards:

- The edges between nodes do not only represent data flow, but also the static nesting of computations in the text that are operating on the data structure. Hence, an `if` statement that controls the execution of a relevant operator appears in the graph as a node connected to the operator node. This allows an `if` statement to be selected and manipulated like other computations.

- Scoping information is added to the tree by connecting each "leaf" operator of the tree to a node representing the containing function. This information identifies paths that not only reference the same data structure, but also originate from the same function. This double relationship may imply that two paths be extracted together as a single new function.

The star diagram provides additional hints to the tool user about the program and the types of operations that can be performed on a given node. Function call nodes have thick borders to highlight that related computations may be hidden in the function called, perhaps leading to an inlining transformation of some sort. Edges in the star diagram denoting flow through a temporary are highlighted in purple to indicate that the computations are not actually nested. Because function definition nodes represent a different kind of node than other star diagram nodes (providing context rather than calculations), we color the nodes blue to make them less obvious, and draw the node as a parallelogram to stress the different meaning. Because the location of a parameter within an operation often affects how the operation affects the parameter, node labels for built-in operations such as `set!` and `if` indicate the parameter position of the contained expression with square brackets. The tool user can thus distinguish cases where a computation occurs in the “test” clause of an `if` from an occurrence in the “else” clause. This feature could be extended to user-defined function calls.

We are also experimenting with other customizations to expose additional relationships. For example, the tool user could ask that assigning the value “0” be distinguished from assigning other values. Such a customization would make initialization operations identifiable directly from the star diagram.

II.C.3 Restructuring Operations in the Star Diagram

The star diagram not only visualizes software structure, but also permits the program to be restructured from the star diagram view.

The star diagram supports six transformations, each making assumptions about the tool user’s intent in order to reduce the amount of information requested: “Extract Function”, “Inline Function”, “Extract Parameter”, “Inline Parameter”, “Rename Function”, and “Make Call”. We also added two functions that built the list of interface functions for the restructuring tool: “Move into Interface” and “Remove from Interface”.

Extract Function. Extracting a computation (or a set of similar computations) makes the associated code the body of a new function, and replaces the occurrences of the computations with calls to the function. (Figure I.1 shows the effect of the transformation.) This transformation resembles the text-based restructuring tool’s extract function transformation, but infers parameters to the transformation such as the destination location of the function. The tool user provides a name for the function; the tool can infer required parameters, but lets the tool user customize them to get the exact interface desired.

In the star diagram, a path from a node to the root denotes a computation on the data structure, and so selecting a node for extraction creates a new function encapsulating the computation on the data structure. If a stack of nodes is selected, then all of the computations denoted are extractable as a single function definition. The tool user can then opt to exclude any of the computations from the extraction, if necessary. Unlike selections in the text view, a selected node can denote a non-contiguous section of code, so the transformation may be required to "gather" statements together with a move transformation before extracting them.

Extract Function places the newly created function at the top level immediately after the variable declaration; if it cannot place the function there, it tries moving the function to other locations.

Inline Function. Inlining a function call replaces the call with the body of the function. Inlining is the opposite of extracting a function; when a function definition inappropriately encapsulates a computation, a possible solution is to inline the function at one or more call sites.

In the star diagram, inlining on a node representing a call replaces the node with the portions of the body that use the data structure.

Inline Function normally inlines only a single call when the transformation is applied to a function call node. It destroys the function and inlines all instances when the transformation is performed on a function definition node.

Extract Parameter. Extracting an expression from a function body (1) adds a parameter to the function definition, (2) replaces the expression with a reference to the new parameter, and (3) changes all calls on that function to pass the expression's result as an argument.

In the star diagram, such an extracted expression always contains a reference to the data structure being encapsulated. Consequently, this operation moves the context of the data structure computation from its current function body to the function's callers. In effect, the operation moves uses of the variable "up" the function call hierarchy of the program.

Inline Parameter. Removing a parameter from a function definition results in replacing the references to the parameter with the expression that was passed as an argument. If there are multiple calls on the function, all the arguments for the parameter must be the same expression. Figure I.2 demonstrates the effect of the transformation.

In the star diagram, the argument expression being inlined always contains a reference to the data structure being encapsulated. Hence removing the parameter has the effect of "pushing" the data structure reference inside the function body and down the call tree so that the function body now encapsulates it.

Rename Function. Renaming a function definition or function call node changes the name wherever the function is referenced. The transformation allows the star diagram tool user to correct inappropriately-named functions.

Make Call. This transformation allows the programmer to convert an expression that matches an already-created function into a call to the similar function. The transformation replaces the expression selected with a call to the function, and identifies what subexpressions become parameters to the function call.

When the tool user extracts a function before all similar expressions have been exposed in the star diagram, the Make Call transformation allows the programmer to replace the single occurrence.

To perform the transformation, the tool user selects the expression resembling the function, then selects a function from the list of interface functions. The tool user then presses the Make Call button.

Move into Interface. Moving a function definition into a module's scope exports the function if it is referenced outside the module's scope, removes the uses of the variable in the function from the star diagram, and adds the function's name to the list of interface functions. As a consequence, only unencapsulated computations on the data structure remain in the diagram, improving the tool user's ability to focus on the remaining portion of the encapsulation task and see the interface of the module.

Remove from Interface. The "Remove from Interface" transformation allows a tool user to remove a function from the module interface if it later is identified as an inappropriate choice. In the star diagram, this has the additional effect of putting the pertinent paths into the star diagram. This transformation is applied by clicking on the Remove button above the list of interface functions.

II.C.4 Choice of Transformations

The star diagram uses a slightly different set of transformations than the text-based restructuring tool because the star diagram is designed for a specific kind of restructuring action (encapsulation), and only presents certain computations and programming language constructs. Each transformation also infers some of the details of the transformation from the code to be manipulated and from knowledge of the encapsulation task. We restrict the number of transformations to minimize the number of actions the programmer must choose from.

We select the transformations to match the actions most important to a tool user encapsulating a variable. Most important would be transformations for creating new abstractions, such as creating functions and destroying function abstractions. The "Extract Parameter" and "Inline Parameter" transformations allow the programmer to change where variable references occur in the calling hierarchy of the program in order to create appropriate functions. The functions for destroying existing abstractions, such as functions and local variable bindings, also allow the tool user to destroy abstractions when the abstractions hide calculations the tool user wants to manipulate. The "Move Into Function" and "Remove from Interface" transformations primarily help the programmer identify progress within the encapsulation.

Watching programmers use the star diagram also identified a number of other needed transformations. We found tool users needed the "Rename Function" transformation when the function's current name does not adequately denote the function's behavior. We also added the "Make Call" transformation to allow tool users to replace common expressions exposed only after the corresponding function was already created.

Other transformations from the text-based restructuring tool do not appear in the star diagram because the operations are irrelevant—the programming language constructs the transformation manipulates are not visible in the star diagram. For example, the star diagram does not display `let` scope constructs, and so the transformations for manipulating `let` scopes are ignored.

II.C.5 Direct Manipulation of the Star Diagram.

To allow manipulating the program through the star diagram, the tool user uses a direct-manipulation interface [Hutchins et al. 85, Schneiderman 83] to specify a restructuring change.

The direct manipulation interface supports the two tasks that the tool user performs frequently: navigating from the star diagram to the source code to identify details in the code that are not visible in the star diagram, and performing restructuring transformations.

To support easy navigation to other views of the program, the tool user can double-click on a node to display and highlight the corresponding text in the program text view. (Changing the selection in the star diagram also highlights corresponding representations of the expression in other graphical views of the program.) When multiple statements are associated with a node, the first is shown immediately in the text view; the tool user can step through the rest of the occurrences in sequence.

To help the tool user quickly identify and perform likely transformations, the star diagram explicitly lists the transformations in a button bar just above the star diagram graph. When the tool user selects a node in the star diagram, the restructuring tool highlights the buttons corresponding to transformations that can be applied to the selected expression, and grays out the buttons that cannot be used in the situation. Because the transformations each operate only on certain types of programming constructs, this feedback helps the tool user identify what transformations can be performed. Less frequently used actions appear in the menus along the top of the window.

The star diagram's direct manipulation user interface also helps the tool user measure progress in the task by displaying what parts of the encapsulation have been completed and what has yet to be done. The star diagram lists the interface functions for the ADT being created in a scrolling list box on the left hand side of the star diagram window. By examining the list of functions encapsulating variable uses and the star diagram's view of the unencapsulated uses, the tool user can immediately identify what parts of the encapsulation task have been completed and what variable uses must still be encapsulated.

II.D Sample Encapsulation of KWIC Using the Star Diagram

For the star diagram to be successful, the tool user must be able to apply the transformations required for meaning-preserving data encapsulation and have enough information displayed to make good design decisions. However, the amount of displayed information cannot be overwhelming.

In order to show how the star diagram tool user performs encapsulation with the star diagram, we can demonstrate the suggested encapsulation on the `*line-storage*` variable in the KWIC Index program. As described in Section II.B, the tool user wants to encapsulate the `*line-storage*` variable in order to simplify modifications of the data structure that affect how the input text file is accessed. This process is a classic data encapsulation, and so the tool user should be able to perform the restructuring using the star diagram.

Exposing indirect uses of `*line-storage*` To begin encapsulating the list, the tool user creates a star diagram for `*line-storage*`. (Figure II.3).

The star diagram shows that `*line-storage*` occurs inside an assignment, `cons`, call to the user-defined function `allwords`, `length`, `list-ref`, and two user-defined functions `csline<=` and `csline`. The user defined functions do not suggest any meaningful abstract operations on `*line-storage*`, so the tool user clicks on `csline<=` to see the related code. The source code shows that `csline<=` takes `*line-storage*` as a parameter. Investigating more, the tool user finds that `csline<=` passes `*line-storage*` into other user-defined functions. This suggests that `*line-storage*` is occurring at too high a level of abstraction. To move `*line-storage*` closer to its uses and expose how `*line-storage*` is directly used, the tool user wants `*line-storage*` to occur inside the `csline<=` function, and inside other functions that merely pass the array down the calling hierarchy. This transformation, inline-parameter, removes a parameter from a function definition, and replaces occurrences of the parameter with the expression being passed into the function. This function only works if all calls to the function take the same parameter, or (as in this case) only one call exists. The tool user selects the `csline<=` node which represents the call to the function, and presses the Inline Parameter button in the button bar at the top of the star diagram window. The restructuring tool checks that the transformation can be performed, changes the source code as the user requests, then redraws the source code window and the star diagram. The resulting star diagram is visible in Figure II.5.

The tool user may see that the star diagram has increased in size. This might be worrisome, but by clicking on the new nodes in the star diagram, such as the `csword` node, the tool user sees that the star diagram is merely showing that the single use of `*line-storage*` in the call to `csline<=` actually represented multiple uses of the variable within the function. However, all the exposed uses of `*line-storage*` appear in calls to `csword` and `cswords`. These functions also take `*line-storage*` as a parameter. The tool user may decide that these functions also represent too high-level an abstraction, and so performs inline-parameter again on the star diagram nodes representing the calls to `csword`, `cswords`, and `csline`. The star diagram after these transformations appears in Figure II.6.

Extracting two line-oriented operations. At this point, the star diagram suggests a likely interface. Following the bottom path in the star diagram starting from the far left, the tool user can see that `*line-storage*` occurs frequently inside a list reference (i.e., `list-ref`), and these instances can be further subdivided into identifying the length of an element via `length` and accessing a member of the list element by another `list-ref`.

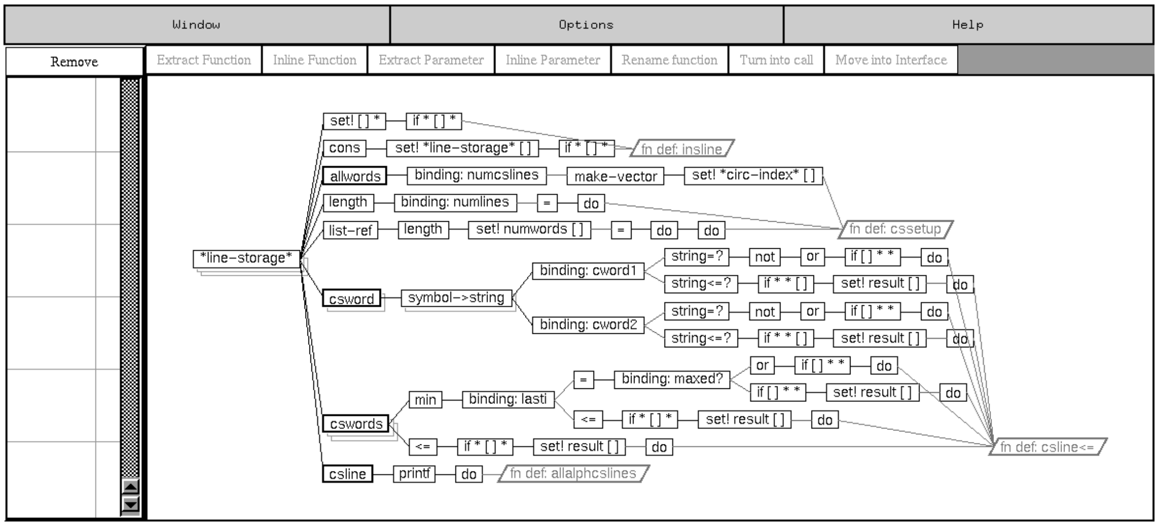


Figure II.5: After performing a push down on functions taking `*line-storage*` as a parameter.

The stacking of the nodes attracts the tool user's attention because these may serve as the implementations of frequently-called interface operations. In the former case, the name of the operation `list-ref`, as applied to `*line-storage*`, followed by a call to `length`, suggests that perhaps a line is being accessed and then the number of its components (the number of words) is being computed. Likewise, in the latter case, the second `list-ref` connected to the right of (i.e., enclosing) the first `list-ref` suggests that one of the components of the line is being accessed. In addition, the tool user sees that these accesses are contained within functions called `cswords`, `csword`, and `wordcnt`. From this information, the tool user may hypothesize that the operations on `*line-storage*` manipulate words in the file, and thus suggest extracting two operations called `words-in-line` and `get-word-in-line`, respectively. However, before attempting the change, the tool user double clicks on each of the node stacks to cause the text view to scroll to the underlying text. In each case, the tool user's hypotheses about the appearance of the code and likely computation being performed are confirmed.

To extract the first node stack as a new function `words-in-line`, the tool user clicks on the `length` stack and then presses the Extract Function button at the top of the star diagram. The tool textually displays all the instances denoted by the node stack,

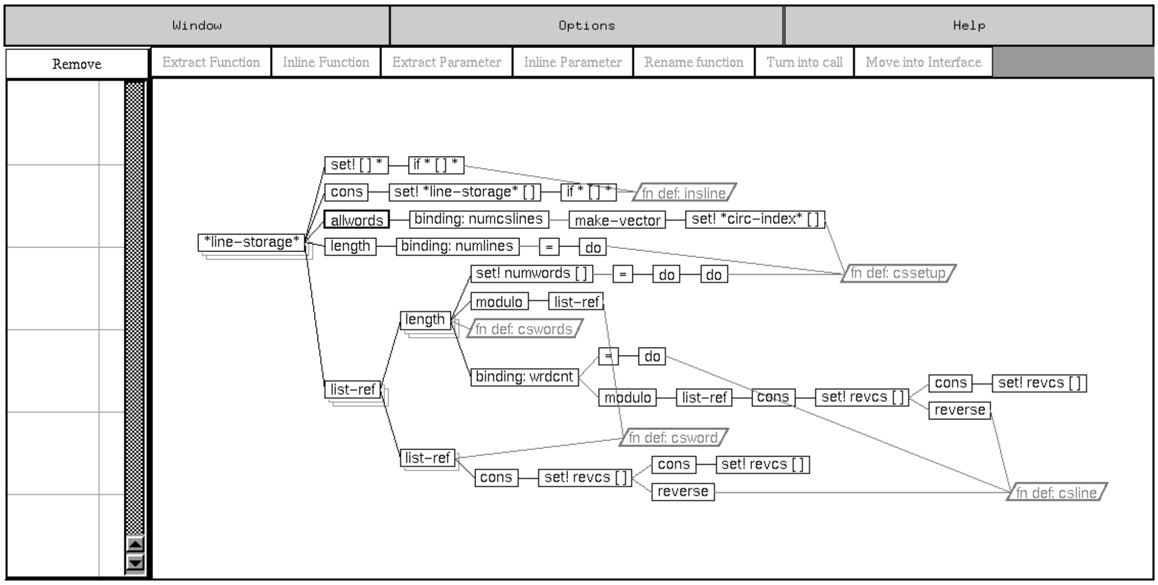


Figure II.6: Star diagram after the push downs are complete.

allowing the user to select which instances are truly representative of the new function. In this case, the tool user decides to replace all of the instances with calls to the new function. The tool then displays a dialog box prompting the tool user for a name for the new function (i.e., `words-in-line`) and for parts of the calculations to be made parameters to the new function (See Figure II.7). In this case, there is one parameter; the variable `line` identifies the line of interest. The tool user fills in these values and a name for the parameter, and clicks on the Extract button in the dialog box. At this point the underlying restructuring tool checks that the transformation will preserving meaning, which the tool finds is true. Then the new function is created and the selected computations are replaced with calls to the new function. Finally, the star diagram is redrawn. The newly extracted nodes are shown on a path to the new function node, and the connections to the right of the `length` node disappear because they now reference `*line-storage*` through function calls. Because the tool user wants this new function to be part of the module interface, the `words-in-line` definition node is selected and the Move into Interface operation is invoked, removing the operation and its path from the star diagram and placing its name on the interface list.

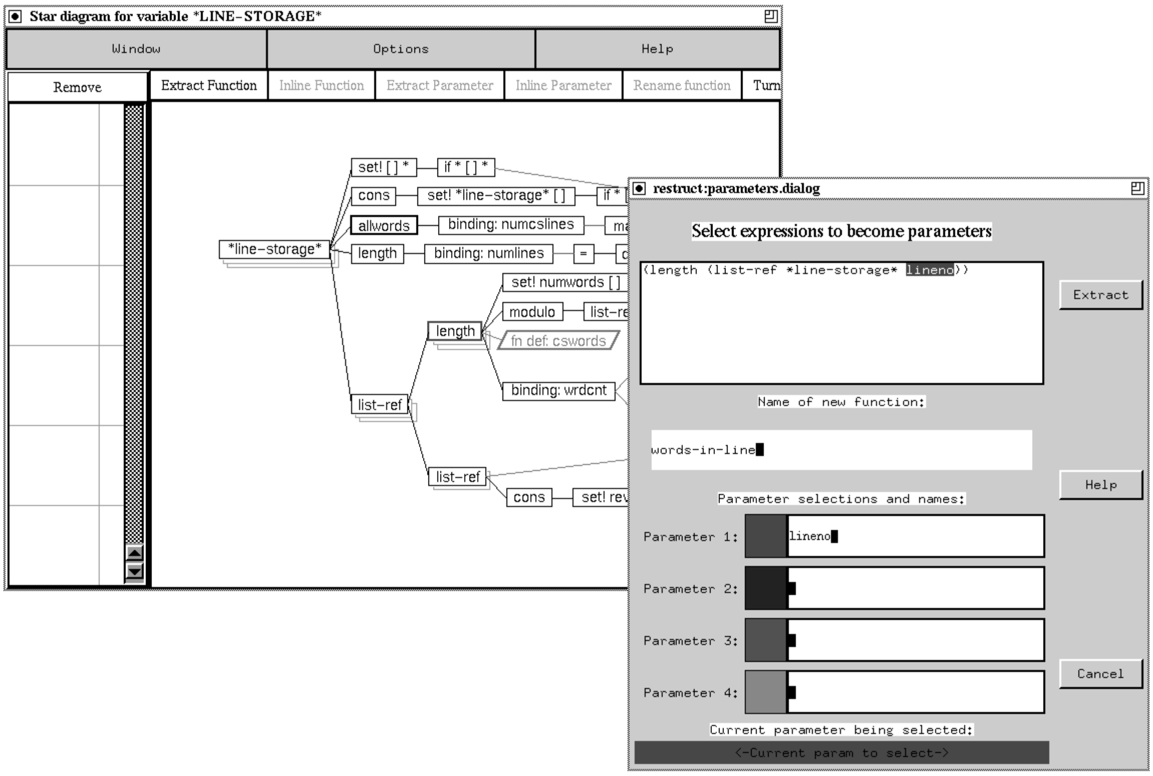


Figure II.7: After selecting the length node in the path `*line-storage*` → `list-ref` → `length` and specifying that all four occurrences should be extracted as a new function, the tool prompts for the new function name and the parameters for the new function (as seen above). Once the parameters and function name are specified, the transformation is performed.

A similar sequence is performed for extracting `get-word-in-line`. Figure II.8 shows the resulting star diagram. The star diagram for `*line-storage*` is now roughly half the size, permitting the tool user to focus on the four remaining unencapsulated instances.

Modifying the interface of an existing operation. The tool user notes that three of these remaining accesses are built-in Scheme operations, and one is a call of a user-defined function (`allwords`). The user-defined call stands out, so the tool user decides to focus on it. It is seen that the result of `allwords` is bound to a variable called `numcslines`, and so the tool user suspects that `allwords` computes a count of all the words in the

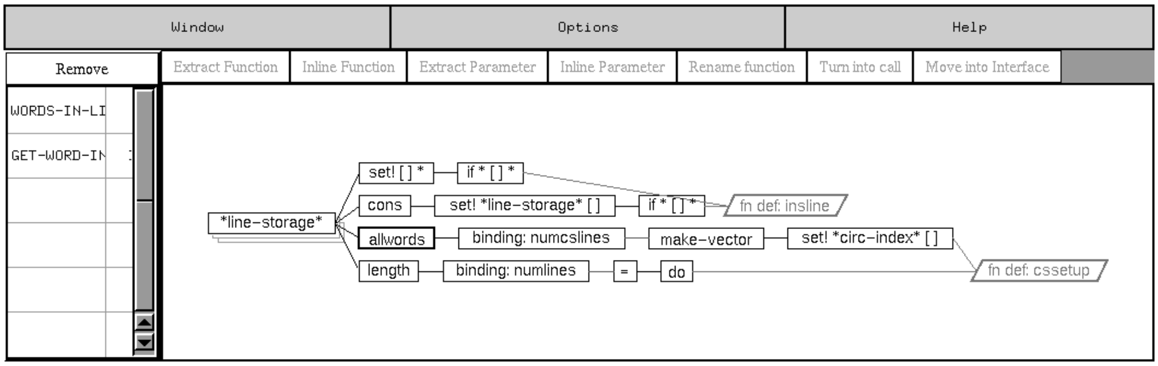


Figure II.8: Star diagram after the functions `get-word-in-line` and `words-in-line` have been extracted.

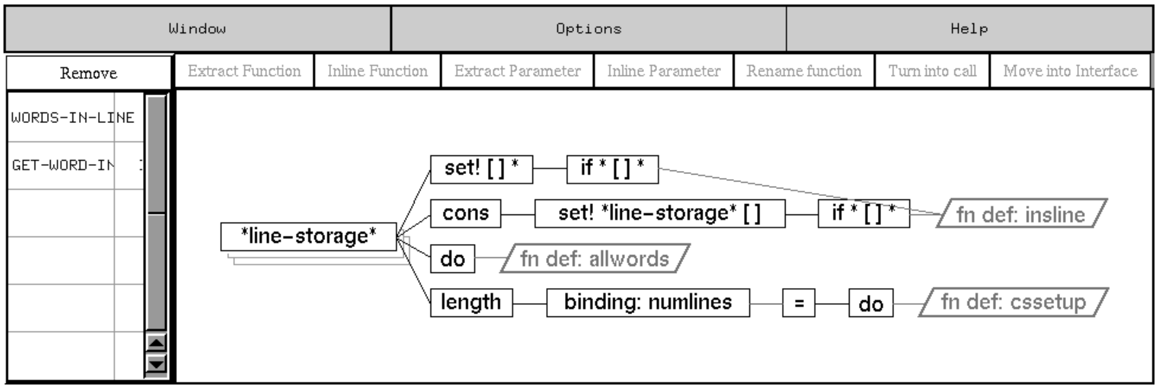


Figure II.9: Star diagram after Inline Parameter has been applied to `allwords`, but before function is placed in the interface.

---

entire line storage. To confirm the hypothesis, the tool user navigates to the code for `allwords`, and sees that it indeed calculates the number of words in the argument list. At this point, the tool user decides that such an operation is a natural member of the interface of the Line Storage module, and hence should encapsulate `*line-storage*` rather than receive it as an argument. The tool user performs this change by applying the Inline Parameter transformation on the `allwords` call node. The tool then checks that all calls on `allwords` pass the same expression, which in this case is trivially true because there is only one call on it. The tool then removes the argument in the call to `allwords`, removes the parameter from the definition, and replaces the use of the parameter with the argument, `*line-storage*`. Finally, the star diagram is redrawn with a path from `*line-storage*` to the `allwords` definition node, replacing the path to the call node. Figure II.9 shows the star diagram after this transformation. Now `allwords` hides a use of `*line-storage*`, so the tool user moves `allwords` into the module interface, which removes its path from the star diagram and puts its name in the interface list.

Completing the encapsulation. The remaining instances are handled similarly to the first two extractions. The `length` reference to `*line-storage*` is verified to compute the number of lines in the list, and it is extracted as `lines-in-file`. The remaining two references appear inside the `insline` definition, which is determined to add words to the list during initialization. The tool user decides that the `insline` function belongs in the interface of the Line Storage module, and moves it into the module interface. This last transformation leaves the star diagram empty, indicating that the `*line-storage*` data structure is fully encapsulated in the Line Storage module by the functions in the interface list: `lines-in-file`, `words-in-line`, `get-word-in-line`, `allwords`, and `insline`.

Because of the resulting encapsulation, the desired enhancement—support for putting lines on secondary storage—can be performed by changing only functions in the Line Storage module. `insline`, which reads the text file into memory, can precompute numbers we will need in other functions, such as the number of words in the file, the number of lines in the file, and the position of beginning of lines in the file. `allwords` and `lines-in-file` can simply return a precomputed value. `words-in-line` can read the requested line from disk and count the number of characters. `get-word-on-line` can read the given line from disk, then return the requested word. These changes can be made with little knowledge of the rest of the program, and therefore may be easier to apply, reason about, and test. In this manner, the tool user can restructure a program in order to simplify a planned modification to a program.

II.E Discussion

The KWIC example shows that the star diagram visualization provides support planning an encapsulation, maintaining knowledge of the current state of the transformation, and identifying what actions are possible at any given time. The example also shows how the star diagram satisfies our design criteria. In particular:

- **Provide global, detailed view:** The tool user sees all of the uses of the data structure in a single, condensed view, instead of having to read through the entire source code.

- **Elide unimportant details:** The details of the exact operation of the code are normally invisible to the tool user, but can be viewed by clicking on a star diagram node. When restructuring decisions require understanding a particular statement in detail, the tool displays the appropriate text. For example, when selecting parameters for a new function, the dialog box for selecting parameters displays the relevant line.

- **Group similar statements:** Contextual information in the diagram helps identify properties that will drive the design of the abstract interface. Stacks of nodes highlight frequently occurring computations that are candidates for becoming part of the ADT's interface. Also, references to the data structure by a function call from a different level than the other operations in the diagram (`allwords`) indicates an inconsistency in the existing interface. The names of the objects in the star diagram provide additional informal knowledge about the program, and can allow the programmer to guess at the behavior of a given statement while looking at the "big picture" provided by the star diagram.

- **Permit restructuring:** All the needed transformations could be applied through the star diagram.

In addition, the star diagram also presents a less complex face of the restructuring tool to the tool user. Because the star diagram's task is tailored to encapsulating a particular data structure, only restructuring transformations relevant to the task need to be available, and some parameters to these transformations can be inferred. For example, the location for a newly created function can automatically be determined based on the location of the module being created. In contrast, the text-based tool forces the tool user to specify all the details of a restructuring transformation because the tool has little idea of the intent of the tool user, and because the tool must be able to perform any of the possible transformations.

The example also demonstrates that the star diagram accrues several benefits by being implemented on top of the existing meaning-preserving restructuring tool. In particular:

- The meaning-preserving nature of the tool guaranteed that errors could not be introduced by the restructuring transformations, freeing both the view and the tool user to focus on the encapsulation task.
- All views in the system—source code, structure diagram, and star diagram—are updated after every transformation. Consequently, the tool user has immediate feedback on the progress of the encapsulation and can always see the current structure of the program to plan the next restructuring action.
- The availability of other views through navigation from the star diagram allows the star diagram to be specialized for encapsulation without compromising other restructuring tasks.

II.F Justification for Features of the Star Diagram

The star diagram's design includes many tradeoffs that at first may appear to be arbitrary. We discuss the rules for generating the star diagram visualization itself in Chapter III; however, we discuss some of the high level design issues in this section.

II.F.1 Separating Nodes Referring to Same Code

In the KWIC and `bussim` examples, separate nodes in the star diagram sometimes correspond to the same code in the program. In `bussim` (Figure I.9), the node

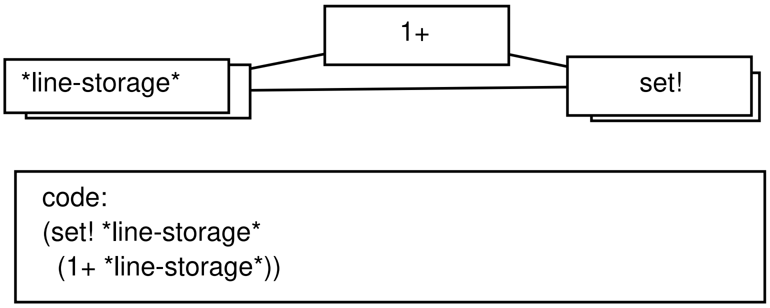


Figure II.10: Sample star diagram where star arms are allowed to converge. With such a scheme, the star diagram would not be able combine `set!` nodes without the `(1+ *line-storage*)` expression on the right hand side of the `set!`, and as a result, would not be able to identify all matching expressions with a single selection.

"`set! *route-distance* []`" at the top of the star diagram refers to the same code as one of the nodes in the stack labeled "`set! [] *`". In the KWIC star diagram (Figure II.6), the `if` nodes in the top two "arms" of the star diagram both correspond to the same if statement. If both nodes represent the same code, why are we not combining them?

This design decision illustrates the differences between a visualization for understanding and a visualization for restructuring. If two arms of the star diagram merged when nodes became similar, the tool user may no longer be able to select sets of functions to execute. For example, Figure II.10 demonstrates a star diagram that combines nodes that represent the same code. In this case, if the programmer selected the `set!` node and pressed extract, the restructuring tool would not be able to decide whether the user was trying to parameterize the left hand side of the set (creating a function for setting the value of *line-storage*) or the right hand side (creating a function that would take a variable and set its value). When the combined nodes only represent a single expression in the code, the programmer can easily specify the subexpressions that will become parameters. However, when multiple, similar statements occur, combining nodes corresponding to the same statement could force the star diagram to represent similar expressions in separate nodes. Because the star diagram is primarily for restructuring, we chose to draw the diagram to simplify manipulation rather than understanding.

II.F.2 Graphical Presentation of the Star Diagram

A second concern for our star diagram implementation is whether we must present this information graphically. For example, if `grep`'s output can show programmers all the uses of a variable, provide the context of each variable use, and elide lines of the program that are irrelevant to restructuring, then perhaps a text-based star-diagram approach could support encapsulation using character-based terminals.

We originally considered a textual encapsulation tool that would list all the relevant lines of the code, sorted according to similarity. As the tool user performed transformations, the list would be updated automatically. When the tool user created a function in the module interface, variable uses inside the function would disappear, thus showing the user only the unencapsulated references.

A sorted, `grep`-like approach differs from the star diagram in the flexibility of the matching and in presentation. Sorting the output would not allow the tool user to ignore irrelevant symbols. For example, when matching assignment statements, the programmer may only care about matching the right hand side of the assignment, not both the variable being assigned to and the value being assigned. Also, because the text window is only able to present a small number of lines at a time, the relevant information may not fit on a single screen.

Rather than listing every matching line in the program, a tree-like structure similar to the star diagram could group larger and larger expressions in the code while hiding the details of individual expressions. The visualization could provide a list of the major operations being performed on the data structure. A textual approach would need to contain some of the properties that facilitate use of the star diagram:

- support matching expressions when some subexpressions may be uninteresting and irrelevant
- easy to view the most common operations that represent likely abstract operations.
- easy to look at larger expressions (to the right in the graphical star diagram) and at the containing function definition when necessary.

All properties are feasible, and thus star diagrams might be presented textually. However, the graphical presentation of the star diagram is probably superior to the textual approach simply because the tool user can easily change the perspective of the view to focus on specific areas of the star diagram visualization.

II.F.3 Choice of New Graphical Notation for Star Diagram

A final design question concerns our star diagram notation. Because many good software design notations already exist, why must the star diagram use a novel notation?

Many design notations present information more suited to high level design than to restructuring or encapsulation. Specifically, most design notations are intended to help programmers understand a program's overall structure and behavior, and thus provide an overall road map of the code. Details of the program's source code are not present in these designs because the details are irrelevant to the understanding task, and the implementation of functions change too quickly for paper designs to accurately capture the code's current state. As a result, graphical design notations portray conceptual objects at a much higher level than the star diagram.

In previous research, we tested the idea of restructuring through notations used for paper designs. We designed and implemented a direct-manipulation graphical restructuring interface [Griswold & Bowdidge 93] based on structure charts [Stevens et al. 74]. Figure II.11 shows the *structure diagram*, the manipulable visualization we created. The structure diagram reveals the high-level structure of the program and allows the tool user to directly change the structure through manipulations on the view. The structure diagrams proved less useful than expected because of the nature of direct manipulation interfaces: the interface only allows the tool user to manipulate the objects that are visible. As a result, the structure diagram proves very useful for manipulating the named objects in the structure diagram: functions, variables, modules, and variable uses. Changing the scope of a function, for instance, only requires dragging the function to the new scope. However, the programmer cannot "grab" a common expression because expressions and individual statements are not displayed in the structure diagram view, and because the similar but dis-

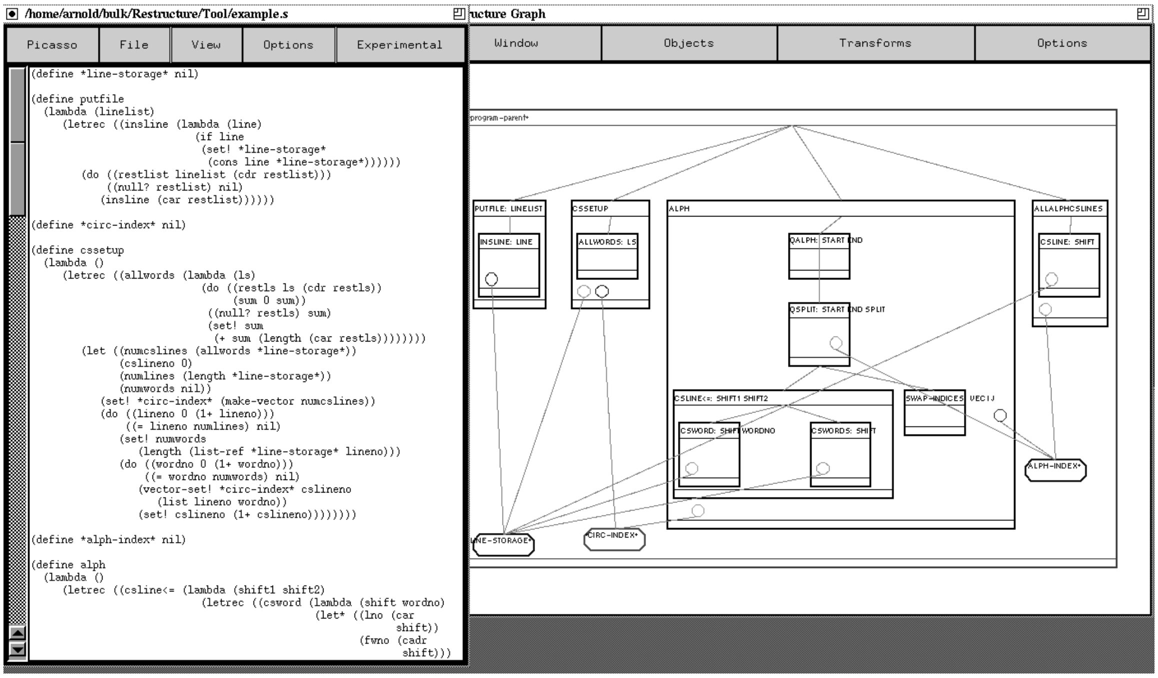


Figure II.11: Text view and structure diagram in the restructuring tool.

tinct expressions would always appear as separate objects in separate scopes. As a result, there is no natural access to the computation around a global variable reference to a data structure. The star diagram's bottom-up view of the uses of a variable displays the code that we want to manipulate.

By comparing the structure diagram and star diagram, we can see that the visualization chosen to help programmers perform a specific action also inhibits other actions. As a result, a useful restructuring environment will most likely need multiple views for restructuring different aspects of a program. In our restructuring tool, the star diagram specifically enables programmers to compare details of scattered source code. The structure diagram, by contrast, proves more useful for manipulating functions, variables and scopes. Other tools might manipulate other aspects of programs, such as the order of functions in the source code to ensure a clean and logical ordering. The inherent number of relations between code fragments in the source code of a program [Brooks 87] makes multiple visualizations of a program necessary.

II.G Summary

The description of the star diagram and of a sample encapsulation describe how the star diagram presents relationships between operations that appear relevant to programmers performing restructuring. Although this example is for a small Scheme program, Chapter V describes how the concept of the star diagram is both applicable to large programs as well as programs written in other languages. In order to test that programmers use the star diagram as we believe, Chapter VI shows that the design criteria accurately fit how programmers actually restructure.

Chapter III

Design and Construction of Star Diagrams

For the star diagram to be usable and practical, the star diagram must be simple to create and draw, especially if we expect the concepts to scale to large programs. The star diagram should also be flexible, allowing the tool user to customize the star diagram's display to highlight pertinent detail for indicating likely restructuring operations, label concepts rather than operations in the code, and hide expressions irrelevant to the encapsulation. Such customizations are necessary so that the star diagram provides as much information specifically related to the encapsulation in the smallest space possible. These requirements—simple, fast, and flexible—drive the star diagram concept.

To illustrate that the star diagram is such an implementation, this chapter describes in detail first how the restructuring tool creates the star diagram. Specifically, we discuss the relationships between similar expressions that the star diagram reveals, then discuss the program representations we can use to calculate these relationships (Section III.A). Second, we describe a generic algorithm for creating star diagrams. The algorithm is generalized to allow the creation of star diagrams for different languages or to customize the information presented (Section III.B). Third, we will present examples of customizing the star diagram view to highlight specific code fragments in the star diagram and condense the star diagram graph (Section III.C). Finally, we detail the current implementation environment used to create star diagrams. (Section III.D).

III.A Representations Needed to Build the Star Diagram

The design of the star diagram is influenced in part by the need to build star diagrams for large programs. The representations of the source code that the star diagram is built from must be inexpensive to compute, and the star diagram itself must also be inexpensive to compute.

The choice of representations is determined primarily by the information we expect the star diagram to display. The star diagram represents relationships between expressions in the program text by two visual cues: stacking of nodes and linking of nodes. Stacking of nodes implies that the underlying expressions may be computationally similar. Linking of nodes implies one expression may be consuming or controlling the other. The star diagram highlights these relationships because they act as informal measures of the cohesion between two statements, and can guide the tool user to place related expressions into the same function or module as part of the encapsulation task. Both can be partially generated from the abstract syntax tree (AST), a parsed representation of the program, with additional dataflow information to identify additional cohesive expressions not identified via the AST.

III.A.1 The Linking Relationship

The first relationship shown by the star diagram—directly linking two star diagrams by drawing an edge in the star diagram between the nodes—connects individual operations that appear to be part of a single computation. Cohesion between nodes (a measure of the "relatedness" of the operations) is likely when one operation uses the result of the previous statement, or when one operation controls execution of another. For example, in the expression

```
(set!  *route-distance* (1+ *route-distance*))
```

the value of `*route-distance*` passes through the increment operator and into the assignment operator, and finally back into the variable itself. This flow of data from one operation to another suggests that the operations are part of a larger computation, and thus may be cohesive. Similarly, in the statement

```
(if (eq count 0) (set!  ptr nil))
```

both the `eq` expression and assignment are controlled by the same `if` clause, and thus probably related.

To identify cohesive operations, we primarily use the syntactic structure of the code found in the abstract syntax tree (AST) to infer both kinds of cohesion. An AST organizes compound expressions or statements as a tree (Figure III.1). Identifying the enclosing expression that uses the result of another can be done by with the AST by moving "up the tree" to identify the enclosing expression. Similarly, the "controlling" relationship can also be captured by one statement being nested inside another, such as an expression being nested within a conditional clause.

"Consumes" and "controls" are, of course, approximate measures of cohesion. However, this metric for cohesion is cheap to generate using the AST. The AST also has minimal space requirements; although an AST can be ten times larger than the source text it is derived from, each star diagram arm only examines expressions within a single function, and thus requires generating the AST for only one function at a time. The AST is also less expensive than other representations that might identify similar information, such as dependency graphs. As a result, following nesting of expressions provides a low-cost method to generate star diagrams.

The nesting does not capture all related expressions because the AST directly represents local relationships only. A producer–consumer relationship also may be introduced through an assignment to a variable, followed by a subsequent reference to that variable. Temporary variables often connect two calculations appearing on separate line, especially in languages, such as C, where style and language structures encourage their use:

a) source code:

```
(list-ref (list-ref ls lno) (modulo (+ i fwno) wrdcnt))
```

b) corresponding abstract syntax tree:

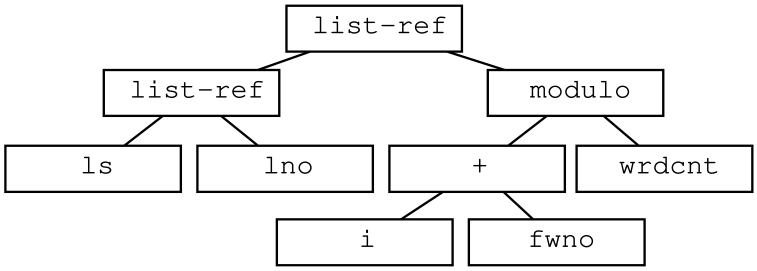


Figure III.1: Sample abstract syntax tree for the Scheme code fragment shown. Note that functions return their value to the enclosing expression, which is represented in the tree as the parent node.

```
int x,y;
x = 0;
y = foo(x);
printf("%d\n",y);
```

In this case, the local variables `x` and `y` connect the initial assignment to `x`, the call to `foo`, and the call to `printf`. When the star diagram algorithm examines the call to `foo`, it sees that the result is saved in a temporary variable, and so identifies the later uses of `y` that can be affected by the assignment to `y`. The star diagram thus identifies `printf` as cohesive with the call to `foo`.

The star diagram algorithm identifies these situations where statements are connected by calculating def-use chains for the variable being set. Although def-use chains are relatively simple to generate [Aho et al. 86] the Scheme implementation extracts def-use information from the program dependence graph (PDG) provided by the restructuring tool.

Using dataflow to identify cohesive expression occurs when the connecting variable is local to the current scope. The star diagram ignores assignments to variables defined outside the local function because the uses may correspond to another abstraction, and because uses of the variable outside the function are likely to represent a different abstract operation on the variable.

III.A.2 Stacking

The star diagram groups similar operations into stacks of nodes in order to identify similarities between groups of statements. The star diagram's stacking of two nodes is based on (1) their predecessors in the diagram being stacked, and (2) their nodes representing the same operation. In this way, the stacking allows the user to identify cases where multiple expressions manipulate the variable of interest in a similar manner.

Both attributes of stacking are easily calculated. The first relationship is known directly from the star diagram. The second can be tested by examining the type of AST operation nodes being placed in the star diagram.

The definition of similarity represented by stacking only implies that the operations, not the arguments, match. However, stronger definitions of similarity could be useful. For example, the star diagram could distinguish between setting a variable to zero and setting a variable to any other value.

III.B Algorithm for Generating a Star Diagram

At a high level, the star diagram algorithm creates the star diagram primarily by examining the AST. (Such methods of deriving behavior of the running code from analysis of the static representation of the code is often referred to as abstract interpretation[Cousot & Cousot 77].) To demonstrate the algorithm, we present a small example. Figure III.2 shows some fragments of code using the `*line-storage*` variable, the corresponding AST, and the resulting star diagram. To generate a star diagram, the algorithm first parses the source code and generates the abstract syntax tree. Next, the algorithm finds each use of `*line-storage*` in the program, and moves from the AST node representing the use of the variable towards the root of the AST. When it reaches the root of the AST, one arm, or path, through the star diagram, has been completed. The algorithm combines similar prefixes from each new arm of the star diagram with the existing star diagram to generate the stacks of similar expressions.

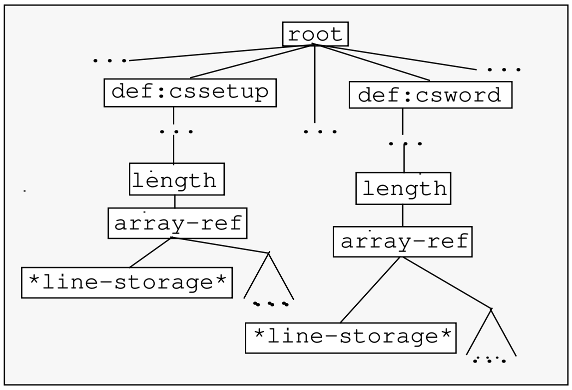
Our detailed algorithm, presented in Figure III.3, extends this description of the

a) source code

```
void cssetup() {
 ...
 numwords=length(lineStorage[line]);
 ...
}

csword () {
 ...
 wordNumber = wordNumber % length(lineStorage[line]);
 ...
}
```

b) abstract syntax tree



c) resulting star diagram

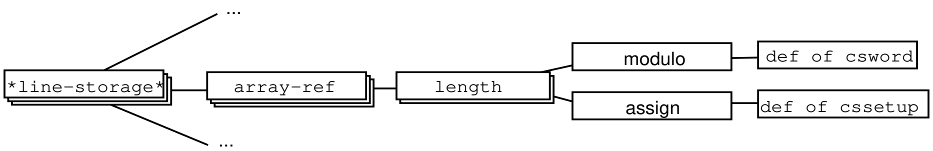


Figure III.2: Generating star diagrams from the abstract syntax tree. The top picture shows a sample fragment of code. The picture below it provides parts of the AST representing that code. The bottom picture illustrates part of the resulting star diagram. To generate the star diagram, we start by finding every use of a given variable (in this case, `*line-storage*`), then move up the tree, combining similar nodes into stacks of nodes in the star diagram.

star diagram by including details such as how the algorithm generates labels, what kinds of nodes in the AST get displayed in the star diagram, and so on. In this algorithm, five operations are parameterized by calls to six customizable functions: **Roots**, **Successors**, **EndNodeTest**, **SkipNodeTest**, **SimilarNodeTest**, and **Label**.

These functions represent the components of the star diagram algorithm. By changing the implementation of the functions, the appearance of the star diagram and the details that it displays can be customized to support the star diagram user's wishes. We describe each function in turn.

III.B.1 Roots Function

The Roots function returns the list of AST nodes that the root of the star diagram will represent.

In the Scheme star diagrams, the root of the star diagram is defined as representing all uses of a variable throughout the program, and thus the star diagram shows the computations consuming the values in the root. To generate the star diagram, the tool user selects any use of the variable. The star diagram generator finds the definition of the variable, then finds all uses of that definition.

III.B.2 Successor Function

The **Successor** function is a heuristic based on our expectations of what counts as the "consumer" of a given operation's result, or what statements should be considered cohesive with respect to the ADT being created. We calculate the successor function primarily from the AST, although the algorithm could also identify consumers using dataflow relations in the PDG. Given a node in the AST, the set of successor AST nodes indicates calculations that consume the result of the expression represented by the node. When no consumer can be found, the successor function returns the AST node representing the syntactic scope in which the computation exists.

The Scheme star diagram algorithm generates successors differently depending

```
DrawStar()
{
    graph = GenerateStar(Roots()); DrawGraph(graph);
}

    /* Given a star diagram node, GenerateStar() calculates the children
        of the given star diagram node, then calls itself recursively for
        each child. */
void GenerateStar(starDiagramNode)
{
    astNodes = starDiagramNode→astNodes; /* expressions represented by node */
    /* Remove AST nodes that represent the end of a star arm. */
    remove-if (astNodes, EndNodeTest);
    successors = (∀ astNode: astNodes) {
        /* Ignore irrelevant AST nodes when moving up the AST tree. */
        do {
            element = Successor(element);
        } while (SkipNodeTest(element));
    }
    /* Divide the successor AST nodes into groups of the successor
        star diagram nodes using SimilarTest function */
    groups = Partition(successors, SimilarTest);

    /* Each group of successors represents a node in the star diagram */
    (∀ group:groups) {
        /* The first AST node in the group is representative of the type of all */
        if ((ASTNodeType(first(group)) == FunctionDefinition) &&
            (already created the star diagram node for the function def node)) {
                insert(FunctionStarDiagramNode(node),starDiagramNode.children);
            } else {
                allocate newStarDiagramNode;
                newStarDiagramNode.astNodes = group;
                /* Generate the label based on either the group, or
                    in this case, on the first AST node in the group, assuming
                    all AST nodes in the group have the same label. */
                newStarDiagramNode.label = Label(first(group));
                insert(newStarDiagramNode,starDiagramNode.children);
                newStarDiagramNode.parent = starDiagramNode;
            }
            /* Recurse until done. */
            GenerateStar(newStarDiagramNode);
        }
        return starDiagramNode();
}
```

Figure III.3: Pseudocode algorithm for generating star diagrams. The parameterized functions (**Roots**, **Successors**, **SkipNodeTest**, **EndNodeTest**, **SimilarTest**, and **Label**) are described in the text.

AST Node Type	Action
Built-in operations	Return the consuming operation.
User defined functions	Return the consuming operation. If no value returned, return the enclosing scope.
Scope	Return the enclosing scope.
Assignment Local variable binding	If previous expression occurs on left hand side, move to looking at scopes. If previous expression is on right hand side and variable is local to the containing scope, find uses of the variable and continue star diagram from each use.

Table III.1: Behavior of the Successor function in the Scheme implementation of the star diagram.

---

on the type of AST node being examined (Table III.1). For built-in operations and function calls that return a value, we use the containing expression as the successor, assuming that the result of the computation is passed to the containing expression. If a node is an assignment (either a `set!` or a local variable binding in a `let` statement), and the expression containing the variable of interest occurred in the right hand side of the assignment statement, then the algorithm assumes the computation is continued where the assigned variable is used. We use the def-use chains encoded as dependencies in the PDG to map from an assignment of a variable to identify the locations where the value may be consumed. If the node represents an assignment to a global variable, then the consuming calculations are probably scattered throughout the code and so do not represent a single local calculation, and again returns the enclosing scope. The algorithm does not display these uses in the star diagram. If the variable being examined in the star diagram occurs on the left-hand side of the assignment, then the star diagram ignores finding other uses of the variable.

If we were to create star diagrams for other languages, the Successor function may have to perform less or more dataflow analysis to identify expressions consuming a result. For example, in a modern language such as C, expression nesting exists, and so we can once again use the AST to infer dataflow. However, creating star diagrams for assembly language would require us to perform some sort of dataflow analysis to identify where a value is used, since the simple grammar of an assembly language provides no method for nesting complex calculations.

III.B.3 Skip Node Test

The successor function ignores certain nodes that are irrelevant to encapsulation and should not be displayed. Such nodes are identified by the **Skip Node Test** functions. AST nodes that are ignored fall into two categories. Some nodes in the AST represent syntactic constructs such as declarations which are irrelevant to the "consumes" or "controls" relation. For example, the ParamList type is a list containing the nodes representing parameters to a function. Other nodes always occur together and would be extraneous in the star diagram. For instance, the lambda function node (indicating a lambda expression) usually occurs directly within the define node corresponding to binding the function to the function's name. Such nodes can automatically be ignored when creating the star diagram. few anonymous lambdas appeared in our sample programs.)

III.B.4 End Node Test

The algorithm stops generating an arm of the star diagram when the AST node being examined indicates that retrieving its successor node will not present any additional interesting nodes. In Scheme, the **EndNodeTest** returns true when a node represents all the code in a function.

III.B.5 Similar Test

The **Similar Test** indicates when to combine nodes representing similar constructs in the program.

In Scheme, two nodes are usually stacked if the nodes' AST node types match. Variable uses and function call nodes require a stricter test; we check the name associated with the node to determine whether the accesses refer to the same variable or function.

AST node	Label description
variable	Print name of variable.
built-in operation	Print name of operation.
function call	Print name of function.
set!	For expression on LHS, “set! [] *.”. For expression on RHS, include variable being assigned: “set! var []”.
do loop	Label indicates if expression occurred in index, test for exit, or body of do.
exitbody (for do)	Indicate whether in test or in result of loop.
if	Indicate whether in test, then, or else clause.
define	Print “fn def:” followed by function-name.
cond	Print “in condition”.
binding	Include name of variable being bound.

Table III.2: Behavior of Label function in Scheme star diagram.

III.B.6 Label Function

The label for a node should help the tool user quickly guess at the behavior of the expression. In Scheme, a label usually contains the operation, such as “plus”, or “call: *function-name*”. In function calls and operations where the parameter that the expression occurs in may convey some knowledge about the context of the use, the label indicates placement of the enclosed expression using the variable. Specifically, we identify parameters in the `set!` operation, and in `if` and `do` loops in the Scheme restructuring tool. In functions where parameter order is unimportant, such as in the + operation, the label function disregards location of the expression. Although the current implementation does not identify the location of parameters for user-defined functions, such a feature could help the tool user already familiar with the code understand the context of a specific node without examining the source code.

Table III.2 describes the behavior of the label function for the Scheme star diagram.

III.C Allowing the Tool User to Customize the Star Diagram

Because a tool user's focus of attention within a block of code depends on the context of the problem being solved, a production version of the star diagram should allow the tool user to control which statements are considered relevant or important to a specific restructuring task. The star diagram generator's parameterized functions permits including such customizations in a star diagram tool.

For example, the statement

```
(set!  *counter* 0)
```

within a program can signify more than simply an assignment to a variable, but represent initialization or resetting the count. A star diagram user might want the star diagram to identify and separately group assignments where the right-hand side of the assignment is zero and where the right hand side is any other value.

By changing the **Similar Test** function, the star diagram user could instruct the star diagram to check the right hand side of assignments, and only match cases of assignment of a variable to zero when either both assignments are setting a value to zero, or both are setting the variable to another value. The **Label** function could then be changed to identify the distinct nature of the two types of assignments.

Customizations could also be designed to help understand a given system by recognizing design or programming conventions that indicate design-level properties of the system being modified. For instance, when remote procedure calls can be detected by naming conventions or analysis of the code, the star diagram could display such calls in a different color to warn that the function call crosses machine boundaries, and thus cannot be inlined. This feature could be added by changing the label function.

Finally, customizations could identify language features. For example, in a star diagram for C, a star diagram user may wish to provide shorthand labels for combinations of common language constructs. The **Label** and **Similar Test** functions could check equalities to see if the right hand side were NULL, and name the node "Null?" rather than specifying only the operation (“==”). We will discuss adapting the star diagram to other languages in section V.

III.D Implementation Details

The code for generating and manipulating the star diagram consists of about 2,000 lines of Common Lisp code. The direct manipulation subsystem is implemented with the PICASSO user interface toolkit [Rowe et al. 91]. The star diagram graph is laid out using the ISI Grapher [Robins 85]. The ISI Grapher draws attractive directed graphs, but trades off speed for output quality. Because the star diagram is primarily a tree representation, with arms combining only at the far right side of the diagram, the star diagram could be laid out with a tree layout algorithm, then have the function definition nodes added after [Chen ss].

Although the previous descriptions of the star diagram generation algorithm suggest that the tasks of identifying information in the star diagram and building the graph are combined, our implementation actually performs the tasks separately, first finding every use of the variable, then creating a star “arm”, the list of nodes representing an arm in the star diagram, and finally creating the tree or graph representing the star diagram.

III.E Summary

In this chapter, we showed that the star diagram is usable and practical. The star diagram uses cheap methods of analysis to identify cohesive expressions that will be of interest to the star diagram tool user. By reimplementing the parameterized function, the star diagram algorithm can be easily customized, allowing the tool user to highlight details pertinent to a specific encapsulation task.

In Chapter IV, we discuss how the generic algorithm can be adjusted to support similar encapsulation tasks, such as encapsulating all uses of a data type, or encapsulating two variables into a single abstract data type.

In Chapter V, we use the generalized algorithm to create star diagrams for other programming languages such as C and MUMPS. Because programs in these languages are much larger than our Scheme example, we require new methods for generating and displaying such star diagrams. In these larger programs, inferring dataflow from the abstract syntax tree means we can generate star diagrams in the time required to parse the source code, and thus make star diagram implementations feasible for large programs. These examples also test how well the star diagram concept scales.

Chapter IV

Limitations and Extensions of the Star Diagram Concept

The star diagrams described in Chapters II and III present a simplified picture of some of the kinds of restructuring actions that the star diagram must support in real programs. First, although the star diagram concept allows us to identify large blocks of similar code that represent a single abstraction, abstractions in real programs could consist of multiple nearby statements in the same scope, or could be represented by many expressions that must somehow be reduced to a single function body. The star diagram requires methods to help programmers create functions in such situations (Section IV.A).

Second, data structures are not necessarily instantiated by a single variable. A data structure may be formed from multiple related data structures (such as a top pointer and counter for a stack), or could represent all variables of a given type (as occurs when creating objects in an object-oriented language). The star diagram must support the encapsulation of multiple related variables, and of all instances of a particular data type (Section IV.B).

Finally, some design decisions, such as identifying that an abstract operation represents too general an operation, cannot be identified from the star diagram. The star diagram can be modified to provide such cues to the tool user (Section IV.C).

IV.A Customizing Selections and Parameters during Function Extraction

One action not adequately addressed in our KWIC example is the customization of selections and the selection of parameters during function extraction. Two problems emerge when we try to identify what code should be in the function and what parameters should be passed as arguments to the function. First, in the case of extracting multiple similar expressions as a single function, we may find it difficult to compare all the expressions to identify similarities and differences, and thus choose the appropriate parameters to the function. Second, identifying the statements that should be part of the function requires understanding what statements are part of the same abstract "operation" on the data structure. Because not all related statements in the program can be identified by nesting and dataflow, the star diagram will not automatically combine all related statements. The programmer thus requires additional support for both determining and selecting related statements.

IV.A.1 Selecting Parameters

To select parameters for functions, we must examine the source code in order to fully understand what the code does and what likely parameters for the new function should be. The initial selection made in the star diagram may include some, but not all, of the code that is desired to be included in the function being extracted. In the simplest case—when there is no node stacking—extensions to the selection can be made by using a text view of the single instance of the code being extracted. However, when multiple similar sequences of code are being extracted as one function definition, a text-based view of code cannot easily display and support manipulation of all the selections simultaneously. As a result, the tool user would have difficulty identifying the appropriate form of the function.

Parameter selection presents a similar problem. The textual view shown in Figure IV.1 uses a prototype instance for the purpose of identifying parameters. Initially the underlying restructuring tool computes the parameters, performing a simple unification on the selections to identify which subcomponents are similar and which are different. Because

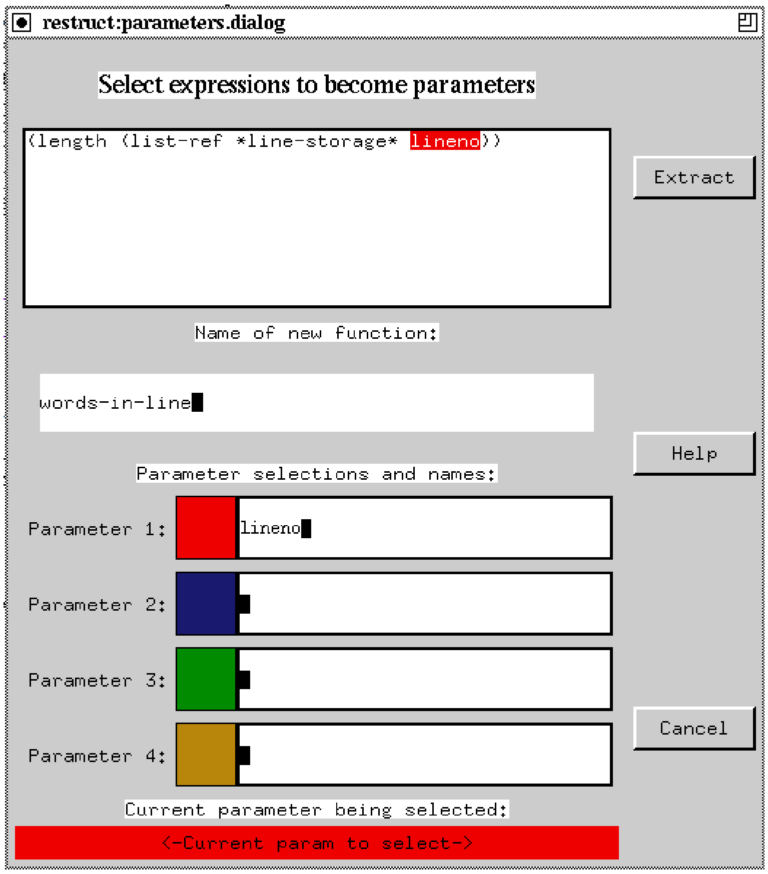
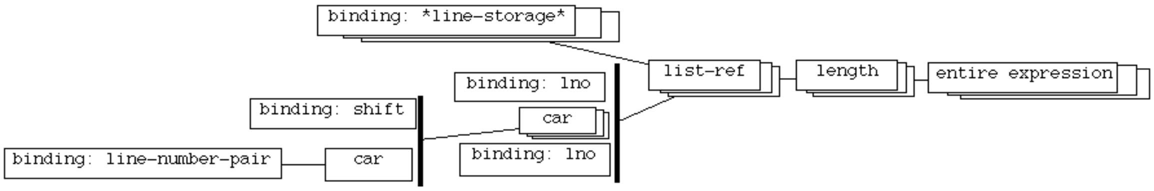


Figure IV.1: Current dialog box for selecting parameters for a new function. The large window at the top displays one of the expressions being converted into calls to the new function; the user selects expressions in the code that will be replaced by a parameter in the new function. This approach will not scale to cases where hundreds of expressions are being converted into a new function because the programmer cannot identify how a given choice of parameters affects the other expressions to be converted.

the function can only include the common aspects of the statements to be executed, the differing portions of the statements must be left outside the function body, and passed as arguments into the function. However, the tool user may identify common portions of the expressions that actually represent calculations that are not part of the abstract operation being instantiated, and would wish those calculations to be passed in as parameters as well. This task requires customizing the default parameter selection made by the tool, which sometimes requires looking at all of the selections to make the best choice, which can become unwieldy.

An alternative approach is to extend the star diagram concept. The current star diagram compares variable uses by examining the enclosing expressions. We could reverse this process to choose parameters, having the arms of the star diagram comparing arguments to the function call. Once the tool user selects a node in the basic star diagram, a separate "backward" star diagram for the selected path can be used by the tool user to extend the code selection and assist in selecting function parameters (See Figure IV.2). A backward star diagram shows all paths corresponding to subexpressions and incoming data flows for the selected code, not just the subexpression associated with the variable being encapsulated. As in the forward star diagram, an edge either denotes nesting of expressions within an argument to the function call, or data flow through a variable assignment to a use in an argument. A backward star diagram for a selected path also overlaps similar nodes. Consequently, the tool user can quickly discern where the similarities of the code selections end. A node with a vertical bar indicates that the given parameter varies between occurrences, with the variants listed next to the bar.

From this view, the tool user can modify the code selection. The default selection corresponds to the portion of the backwards star diagram that was initially selected in the basic star diagram, which includes all similar expressions nested within the right-most node in the initially-selected path. The selection can be extended (or contracted) one path at a time by selecting the node at which to break the path. All nodes to the right of a selected node are included in the selection, and the selected node and all nodes to the left are excluded. By extending the selection to include paths that are not common to all




```
(length (list-ref *line-storage* lno))
(length (list-ref *line-storage (car shift)))
(length (list-ref *line-storage* lno))
(length (list-ref *line-storage*
                  (car (car line-number-pair))))
```

Figure IV.2: A possible backwards star diagram for the path `*line-storage*` → `list-ref` → `length`. The four statements at the bottom are the lines of code represented by the drawing.

---

selections, only the paths common to the extended selection are retained for extraction. For example, in Figure IV.2, the basic selection includes only the top path, because the bottom path immediately splits into two different variable bindings and a call to `car`. Selecting the right-most vertical bar in this view retains the maximal selection—everything to the right is retained in the selection, the selected node and everything to the left is removed. If, on the other hand, the left-most vertical bar is selected, then the `car` node is included on the path to the right, thus eliminating the variable bindings.

Parameter selection can be performed as part of the customization process. When a node is selected for exclusion, the computations excluded must become arguments to the function so that their values can reach the included code.

IV.A.2 Identifying Related Statements

Although the backward star diagram increases the encapsulation choices provided by the basic star diagram, some choices are still prohibited or disguised.

For instance, there might be output statements that pertain to the data structure, but do not actually access it. In KWIC, the following code prints out the final index:

```
(let ((numcslines (length *alph-index*)))
```

```
(printf "Alphabetic listing:~%")    ; excluded
(do ((i 0 (1+ i)))
    ((= i numcslines) nil)
  (printf "~s~%"
          (csline (vector-ref *alph-index* i)
                  *line-storage*)))))
```

No forward or backward star diagram would display the `printf` before the `do` loop, because it is not nested in any operation or linked through assignments to the surrounding code. However, if the tool user could get access to this code through the star diagram, it might reasonably be chosen as part of a new `print-index` function. Comments are likewise a problem. In the case of an input–output operation like `printf`, there is in fact an implicit global variable—the file system—being read and modified, thus "connecting" the two `printf`s. If the data flows to and from this variable were included in the backward star diagram, the first `printf` would be included as an implicit input to the second `printf`. However, this solution only handles relationships corresponding to data flow, prohibiting the tool user from forming modules based on other relationships.

We can consider two methods for including computations not related by data flow or nesting in the star diagram:

1. `let` nodes (a scope declaration that can include definitions for local variables) could be included in the diagram, so that the tool user is aware of extra levels of nesting and can select them. This method assumes that code inside a `let` scope is cohesive. However, including `let` nodes results in cluttered diagrams with some programming styles, and scoping is an imperfect indicator of relatedness.

2. The tool user could customize a selection through the text view, growing it or shrinking it as deemed necessary. To provide some of the benefits of the star diagram in text, we have prototyped a text-like view that highlights forward and backward data flow from a selected piece of code. This view simplifies the selection of data flow extensions of the selected code, without sacrificing the ability to include other nearby statements. As noted earlier, however, the primary problem is that if there are multiple selections, it is sometimes not practical to view and manipulate them all together as text.

Both of the problems presented here show that planning restructuring requires the tool user to understand details of the source code not shown in the star diagram. Planning the parameters for a new function may require the star diagram tool user to compare differences between a large number of ostensibly similar expressions. In other cases, planning a new function may require the tool user to read comments or identify statements related in a manner other than the nesting and dataflow presented in the star diagram. In either case, if the tool user must examine too much code to plan the function, the tool user may be forced to plan an ill-advised function without truly understanding all uses of the variable, and incorrectly all parts of the abstract operation. While the star diagram reduces the amount of code the tool user must examine in order to identify similarities between many expressions, similar methods of reducing the code the tool user must examine for selecting parameters and selecting expressions are required to further extend the star diagram's ability.

These problems are related to the original purpose of the star diagram: how do we reduce the amount of code the programmer must examine? Solutions to these problems will extend the scalability of the star diagram.

IV.B Other Forms of Encapsulation Tasks

Our current star diagram currently supports encapsulating a single data structure instantiated as a variable. Although some programs require the encapsulation of a single global variable, other scenarios include encapsulating a type or encapsulating a data structure made of two variables. The star diagram concept can be extended to handle both cases.

IV.B.1 Encapsulating Types

Our star diagram for Scheme currently supports encapsulating a single data structure, rather than all instances of a data structure. This works well for many, but not all, programs. In KWIC, there is a single data structure for `*line-storage*`, but a different implementation might support several such stores.

Although hiding types rather than individual data structures is somewhat compli-

```
/* C++ code for the TreeNode structure. */
/* Return the number of children of a given node. */

int TreeNode::Children()
{
  if (this->children == NULL) {
    return(0);
  } else {
    int count=0;
    TreeNode *child = this->children;
    while (child != NULL) {
      count++;
      child = child->sibling;
    }
    return(count);
  }
}
```

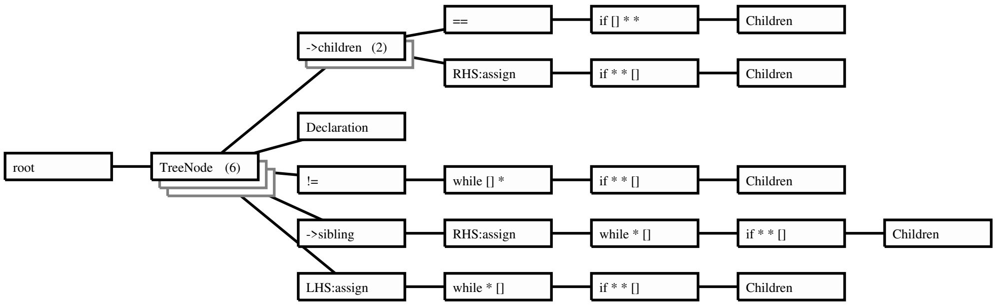


Figure IV.3: Type-oriented star diagram for a fragment of C++ code. The star diagram shows all the references to all variables of type TreeNode in the above source code.

---

cated by the lack of type declarations in Scheme, a typed language such as C provides much of the needed information to support a type-oriented star diagram. The Scheme algorithm for the star diagram searches for instances of the variable(s) that denote the data structure. In a star diagram for C, this approach can be generalized to search for all instances of variables and expressions of the chosen type. When displayed, all the instances are collapsed to a single root node, as in the diagram for Scheme. Figure IV.3 shows an example of a type-oriented star diagram.

The potential shortcoming of finding references based on the type of the representation is that two uses of a common data type might be used for sufficiently different purposes, and are not really instances of the same type abstraction. For example, some abstraction might be represented by variables of type pointer to char (or String), but all variables and expressions of type String might be part of the same abstraction. One way to avoid such false matches is to allow the tool user to selectively exclude instances of the type. In this way, the tool user can perceive how the diagram is affected and decide what instances are appropriate to keep.

Type-oriented encapsulation also results in potentially more informative diagrams. Consider the `allwords` call from the example of encapsulating `*line-storage*` in Section II.D. In that case, only the function call node for `allwords` was shown in the star diagram, but not its body, requiring the tool user to examine the contents of `allwords` with the text viewer to see how `*line-storage*` is manipulated within the function. However, it might have been more helpful to actually see the body as part of the star diagram, because this would reveal overlaps with other computations. With type-oriented encapsulation, both the call and the body are in the diagram because the parameter is of the type of `*line-storage*`. One potential downside is that displaying both might cause size problems.

In a language such as C, type encapsulation could miss operations upon a data structure if the structure is cast to a different type before beginning the operation. By treating typecasts as basic operations on the data structure, however, the casts appear in the star diagram when the object of the examined type is cast to the new type, and can be manipulated like other expressions.

IV.B.2 Pointers to the Data Structure

Not all accesses to a data structure are necessarily through a single variable, as the basic star diagram assumes. In some cases, references to the structure may be saved in another variable, creating an alias to the structure that may be referenced later. Type encapsulation naturally avoids the alias problem by including all references of the type, including references through pointers of that type.

However, even in the current Scheme star diagram, aliasing can be handled. The underlying restructuring tool generates alias information when building the PDG [Larus & Hilfinger 88], so that aliases can be detected and treated as references to the variable being encapsulated when displaying the star diagram. One possible problem with this approach is that the analysis is conservative, so extra aliases may be identified. Another way of thinking about aliases is to note that for an alias to exist, it must first be created by a direct reference to the variable denoting the data structure being encapsulated.[1] Consequently, the calculation introducing the alias will appear in the star diagram and be visible to the tool user. Seeing such an assignment, the tool user can choose to restructure the program to encapsulate the alias, or explicitly fold the alias into the star diagram so paths related to it can be viewed and manipulated.

IV.B.3 Data Structures using Multiple Variables

Related expressions may not be included in the star diagram because the data structure being abstracted is actually represented by multiple variables. For example, a linked list could be represented by a head pointer and a counter for the length of the list. The preferred encapsulation would include functions manipulating both of the variables. To support such an encapsulation, the tool could create overlapped star diagrams for both variables. When the two uses of the two variables are sufficiently intertwined in their usage, their paths will overlap significantly, and a single selection by the tool user could select the appropriate computation. However, in the linked-list example, the update of the counter appears to be largely independent of the head pointer—the two paths merge only at the function definition node that contains them both. This hint, however, could lead the tool user to make two simultaneous path selections—the one for the insert and the one for the counter increment—and inspect the textual relationships of the resulting selections in the text view before choosing to extract them together. This approach could also facilitate combining two references to the same variable into a single extraction.

[1] In the C programming language, however, coercion and pointer arithmetic can disguise an alias.

IV.C Showing Calling Context

A third limitation of the current star diagram is that it occasionally shows too little information to identify cases where destroying existing abstractions, such as inlining a function, could be necessary.

Consider a function definition that may represent too narrow an operation. In the KWIC example, we could have encapsulated `*line-storage*` so that the functions manipulating the array manipulated lines, not words. A key function in such an implementation would be `get-line`, which would appear as:

```
(define get-line (line-number)
  (list-ref *line-storage* line-number))
```

Such a function hides little of the data structure. The rest of the program would understand that lines are lists of words, and there would be significant coupling between `get-line` and its callers with respect to the data structure being encapsulated. As a result, if we choose to reimplement `*line-storage*` with a different format for lines in the array, we would have to again apply a global change to the entire program. In fact, most of the uses of `get-line` would be in two contexts: either getting the number of words on a line, or getting a specific word. If the tool user destroyed the get-line function using the Inline Parameter transformation, such a conclusion would trivial to deduce from the star diagram.

However, the star diagram provides no hints that the function call hides two common calculations that may represent better functions for the module interface, and thus that inlining the function definition would expose these new abstract operations. A tool user examining the star diagram for the `get-line` use would not see that the return value of `get-line` only is used by `length` or another `list-ref`. Instead, the tool user would only see the star diagram arm:

`*line-storage*` → `list-ref` → `defOf:get-line`

The problem is that the current star diagram assumes that the calling context of the function is not important enough to display in the star diagram. In the standard star diagram, paths are terminated at the level of a function definition to keep paths short and ensure that every path is extractable as a function. This approach does not show the contexts in which the function is being called, which can complicate the process of making a decision to inline. The text view also fails to provide this information readily, because the call sites of a function are typically distributed through many files.

To overcome this lack of visibility, the star diagram can be extended naturally to include paths beyond function definitions into their callers. The star diagram could display additional star diagram nodes to the right of the function nodes showing how the function call is used. For the example of `length` being performed on the return value from a call to `get-line`, the tool user would see a star diagram arm of the form:

`*line-storage*` $\rightarrow$ `list-ref` $\rightarrow$ `defOf:get-line` $rightarrow$ `length`

Such a solution requires deciding how many levels of star diagram nodes are provided to the right of the function nodes. Since the utility of these nodes depends on the restructuring goals of the programmer, decisions about displaying additional nodes is probably best left under the control of the tool user.

IV.D Summary

This chapter presents both the strengths and weaknesses of the star diagram. First, although the star diagram presents an elided view of the information needed by the programmer to plan an encapsulation, some operations, such as choosing parameters for a function or deciding which expressions are related, actually require the programmer to understand the code in depth. Thus, the task requires at least some examination of the source code of the program. Returning to the text is simple for small numbers of references, but planning a change for hundreds of uses can cause the programmer to have to examine multiple blocks of source code. Although the star diagram reduces the need to examine code, the programmer must eventually examine the source code for authoritative answers about the purpose and intent of specific code, and to identify exactly the code to manipulate. This need to examine the source code is required because of the richness of the textual representation of the source code, existence of domain-specific information that the star diagram cannot know, and because restructuring a program is intended to support the programmer's maintenance task.

The extensibility of the star diagram concept demonstrates its strength. The examples—encapsulating multiple variables, encapsulating types, and extending star diagram arms across function boundaries—shows the flexibility of the star diagram as a visualization for identifying similarities between code connected by dataflow.

Chapter V

Using the Star Diagram on Real-World Programs

For the star diagram concept to be useful, it must be usable on programs as found in the real world. These programs may be written in languages with more complex syntax than Scheme, such as the C programming language. Real-world programs are also much larger than the typical Scheme program. Programmers will need to understand and encapsulate variables or types occurring hundreds of times in programs measured in the thousands or millions of lines, rather than variables occurring tens of times. Unless star diagrams can be created for programs written in common languages, and used to understand and manipulate large systems, the star diagram concept will be irrelevant in the real world. We can evaluate the applicability of the star diagram concept to other languages by creating star diagram generators for these languages. These star diagrams can then be used to test scalability of the star diagram visualization for large programs.

First, we examine the question of whether star diagrams can be created for other languages (Section V.A). We show the feasibility of star diagrams for other languages by describing how such star diagrams can be created, and describing in detail our implementation of a C star diagram generator. We then present an example of a C star diagram to show that these star diagrams can help a programmer understand a program. Finally, we discuss the language features that complicate creation of star diagrams, and suggest how the star diagram implementation can be adjusted to avoid these problems.

Second, we claimed in previous chapters that the star diagram provides a concise, easily comprehended view of uses of a given variable. Thus, a star diagram presenting occurrences of a frequently used variable in a large program must be reasonably compact for the programmer to be able to use the star diagram for encapsulation. To examine how the star diagram approach scales to large programs with frequently used variables, we construct star diagrams for several programs written in the C and MUMPS programming languages (Section V.B). We examine both infrequently used variables, frequently used variables, and type-oriented star diagrams (introduced in Section IV.B.1). (Table V.1 lists the programs examined, and their relative sizes.) We use measurements of these star diagrams to extrapolate how the star diagram concept scales. We find that although the horizontal size of the star diagram remains small, the vertical size of the star diagram can become excessive for large programs. To solve the difficulties of size, we propose methods for eliding unnecessary star diagram arms (paths through the graph) either by identifying and marking common programming constructs or cutting uninteresting arms.

Our primary intent for creating the star diagrams in this chapter is to test the scalability of the star diagram concept; although these star diagrams contain much of the same detail of the star diagrams presented for Scheme, these star diagrams do not implement all aspects of the algorithm described in Chapter III. For example, because an algorithm for drawing directed graphs was not readily available, we chose to draw star diagrams as trees. As a result, C and MUMPS star diagrams may have multiple function declaration nodes on the right side of the diagram for a given function name.

We also do not display star diagram edges representing dataflow from the assignment of a temporary variable to its uses. Section V.C will discuss why these limitations should not affect the validity of the star diagram size figures given.

program	LOC	variable/type	variable references	size of `grep` output (lines)
KWIC	200	`*line-storage*`	10	10
Adventure Interpreter	1,400	`currentRoom`	36	34
		`items`	83	78
		`rooms`	25	18
Emacs	76,000	`minibuf_window`	48	65
		`current_buffer`	1909	1,354
		`struct buffer`	2409	N/A
CHCS (MUMPS)	590,000	`DIC`	46,613	43,257
SunOS (UNIX)	88,000	`struct mbuf`	1,035	N/A

Table V.1: Description of the programs and variables examined using star diagrams. All programs are written in C, except for CHCS which is written in MUMPS. Most star diagrams display uses of a variable, except for `struct mbuf` and `struct buffer`, which are type-oriented star diagrams. Because `grep` cannot identify all uses of variables representing pointers to a specific structure, no figure is given for the size of grep output when searching for all uses of a given structure.

V.A Star Diagrams for C Programs

To show that the star diagram concept can be applied to other languages, we demonstrate that star diagrams can be created for programs written in the C programming language.

C star diagrams are an attractive case studies to identify the practicality of star diagrams. Large C programs are common, the language is extremely popular, and the language contains features not seen in Scheme, such as macros and type coercion. We have implemented two types of C star diagram generators: standard star diagrams that assist encapsulation of a variable, and type-oriented star diagrams that assist encapsulation of a type.

We describe how to implement a star diagram generator using the algorithm described in Chapter III, then provide a small example of how a star diagram for a C program appears. Finally, we discuss the language features that make generating star diagrams difficult, such as language grammar and macro processing. We also discuss how the star diagram algorithm can be adjusted to avoid these difficulties and support other languages.

V.A.1 Creating Star Diagrams for C Programs

The AST-based algorithm presented in Chapter III can be used as-is for a C star diagram. C's syntax allows us to identify implicit data and control dependences directly from the AST, which is required for the basic algorithm to work. The only changes to the algorithm are in the language-specific details of the algorithm, parameterized inside the specialized functions. These functions—**Roots**, **Successors**, **TerminatingNodeTest**, **SkipNodeTest**, and **Label**—require slight changes when converting the algorithm to C. The **Roots** function and **SimilarNodeTest** functions both operate as in the Scheme algorithm. The **Roots** function, which finds all roots of a star diagram, finds all uses of a given variable (or, as in Section IV.B.1, all uses of all variables of a given type.) Because our implementations of C star diagrams use only the AST and have no access to dataflow information, the successor function finds the containing expression by always moving up the AST. The **TerminatingNodeTest** merely checks for the root of the AST. The **SkipNodeTest** again excludes AST nodes considered irrelevant to the encapsulation. For example, the C star diagram algorithm ignores label statements because such statements provide few clues about the computation performed. The C grammar used in our AST also represents compound statements and lists of parameters as nodes in the AST, and so our **SkipNodeTest** ignores such nodes.

One significant difference between the C and Scheme star diagrams—performing the **SimilarNodeTest**—is introduced by our implementation of the star diagram generator, which is organized as a set of UNIX filter-style programs. The first program examines the code and creates individual arms for the star diagram; the second matches and groups star arms, then produces adjacency lists for the graph; and the third filter lays out and draws the graph. Because the matching and grouping program does not have access to the AST when stacking nodes in the star diagram, the **SimilarNodeTest** compares labels to decide if two nodes nearby in the tree should be grouped, rather than directly comparing the AST node type.

The type-oriented star diagram generally behaves similarly to the standard star diagram generator. To support type-oriented star diagrams using the star diagram generator, we changed the **SimilarNodeTest** and **Roots** functions. The **Roots** function must find all uses of all variables of the given type, rather than all uses of the variable; the change merely requires searching the AST for all variable declarations and comparing the type declaration to the type of interest. The **SimilarNodeTest** function's behavior changes because the matching function needs to overlap all root nodes, even when the roots represent variables with different names.

Both implementations of C star diagrams generate the star diagrams compactly and quickly. The AST generator we use for creating C star diagrams, created as part of a prototype C restructuring tool [Morgenthaler & Griswold 95], provides the basic infrastructure for creating the star diagram: the ability to traverse the tree; find all uses of a named variable; and find all variables of a given type. If we were to move the star diagram to a new environment, these properties would be the most essential to implement the star diagram.

Star diagrams are drawn using Moen's tree drawing algorithm because of its speed and compact layouts [Moen 90]. The majority of the time required to create the star diagram is due to parsing of the source code. The other phases of the algorithm—finding uses, generating star arms, merging arms, and drawing the graph—add negligible time. For the current implementation, generating star diagrams takes about four seconds to draw a star diagram for a variable with 36 uses in a 1,700 line program, and 300 seconds for a variable with 2000 uses in a 75,000 line program.

V.A.2 Example of a C Star Diagram

The star diagram algorithm gives few clues whether the star diagrams for other languages will be informative. To strengthen our claims about the feasibility of implementing star diagram generators for other languages, we can plan a sample encapsulation from a C star diagram. This example also suggests the appearance of other star diagrams that, for space reasons, cannot be presented in their entirety (Section V.B).

The program we will examine, `adventure`, is a dungeon exploration game [Adams 80]. The game allows the player to move around a virtual world, divided into *rooms*, in order to solve a puzzle. The full description of the puzzle—room descriptions, objects, and results of actions—are stored in a data file that the interpreter reads during initialization. `adventure`, a port of a BASIC program into C, consists of 1,700 lines of code, excluding comments and blank lines.

To motivate restructuring `adventure`, let us imagine that we need to change the `rooms` data structure in `adventure`. `adventure` maintains a global array of room descriptions, called `rooms`, where each array entry is a pointer to a `Rooms` structure. Each `Rooms` structure contains a room description and an array of destination room numbers indicating where the player moves when exiting in of the six possible directions (north, south, east, west, up, down). The game loads all `Rooms` structures into memory during initialization.

We could desire to extend the size of puzzles in the interpreter by creating more rooms than memory allows. Thus, like the `KWIC` example, we would like to cache room data structures on disk, and only read relevant rooms into memory.

To plan this encapsulation, we can examine a star diagram for the `rooms` variable. First, we generate the star diagram for the variable (Figure V.1). This star diagram represents 25 uses of the variable throughout the program. We can see from the star diagram that the `rooms` variable already represents parts of an array abstraction because 20 of 25 accesses occur inside an array reference (represented by the `arrayRef` node), which occurs during a field access (one of the `getField:` nodes). So, even without the source code, we can identify that the `rooms` variable is an array, and it is an array of structures. Of the four remaining uses, one is an assignment to the variable. If we examine the source code associated with the assignment, we find it actually represents the allocation of the array using malloc. using `malloc`. The final four references are represented by the chain of nodes

`PtrDec` → `FullDec` → `top-level`

which represent the external declarations for the variables found in headers included by each file, and thus are uninteresting to our encapsulation.

Most of our attention is focused on the array references because of the number of occurrences (indicated by the stacked nodes). We can see that two nodes are connected to the array reference node, and both represent accesses to fields of a structure. We can surmise the first, the `desc` field, represents the room description because many of the accesses occur within the `DescribeRoom` function. The second, `dir`, is harder to diagnose. We could predict its behavior from a knowledge of the game's behavior, from the scope of some accesses (such as `DoGoAction` which probably handles the player's `MOVE` command), or from comments in the source code. We can see from the star diagram that the `dir` field must be an array, because it always occurs as part of an array reference. If we examine the code, we will find that each entry of the array represents exits from the room. Thus, we could decide at this point that a likely function for the interface of a module encapsulating the `rooms` array could be a "RoomInDirection" function that, given a current room and direction, returns the room number located through that exit. However, we can choose to examine deeper levels of the star diagram to see if other candidate functions exist.

Although the array references occur in what seem like a large number of contexts, we can quickly identify the contexts of interest. The call to `printf` and the `dereference` node within a call to `fscanf` occur in the function `ReadDataFile`, and are actually initialization and debugging code in the routines for loading a puzzle. These can either be replaced by accessor functions, or the initialization function can simply continue to directly access the internals of the `Rooms` module.

The other three contexts are more interesting; by examining the source code, we see that the "`!=`" node represents code testing if the direction out of a room is zero. The code represented by this star diagram node probably instantiates a test for the existence of an exit from the room. The test for equality (represented by the star diagram node directly below the inequality) tests for equality to zero. That test probably checks for an the non-existence of an exit. These represent complementary functions, and so can be created as two functions with a restructuring tool, and replaced after restructuring by a single call

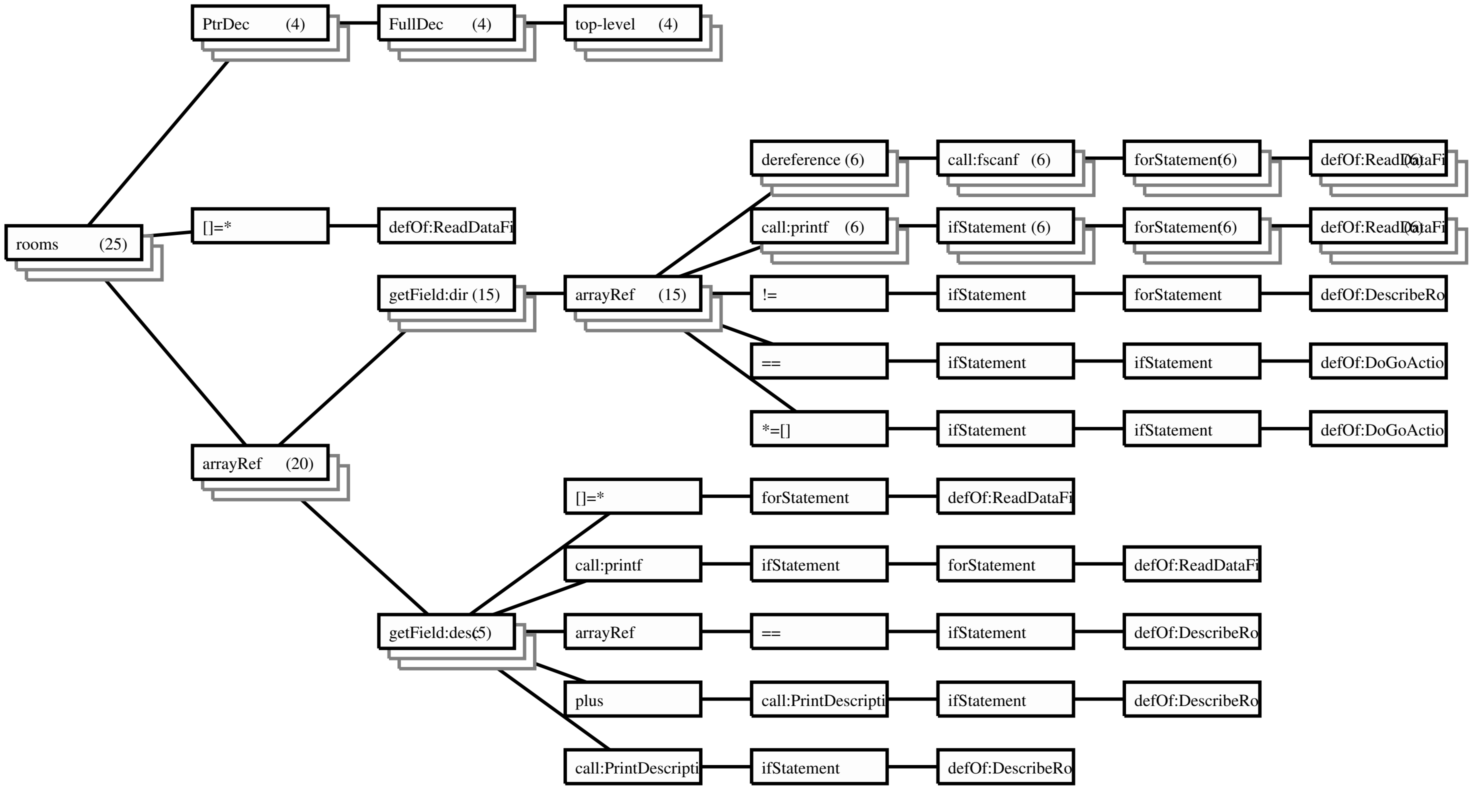


Figure V.1: Star diagram for the `rooms` variable in the Scott Adams Adventure.

using a negate operation on one of the calls. The final case, assignment, appears in the source code to set the `currentRoom` variable to the new room when the player moves.

Uses of the `desc` field can similarly be examined through the star diagram. Three of the five uses occur in the `DescribeRoom` function, which suggests that there are not three simple computations occurring, but one complex operation using the variable in different contexts. We can examine the source code to confirm this hypothesis:

```
if ((rooms[currentRoom].desc)[0] == '*') {
    PrintDescription(rooms[currentRoom].desc+1);
} else {
    PrintDescription("I'm in a ");
    PrintDescription (rooms[currentRoom].desc);
}
```

From reading the source code and comments, we can identify the behavior. This code prints out the description of the room; a star at the beginning of the name suppresses the usual "`I'm in a`" prefix before printing the room description. The `if` statement thus represents the "PrintRoomDescription" abstraction, while the existing `PrintDescription` function actually represents describing the room as a whole—the items in the room, the room's description, and other details of the setting. This `if` statement occurs in three of the five branches off the `desc` node in the star diagram. The other two uses, the assignment to the description field and the `printf`, again perform initialization and debugging in the initialization function.

We can thus use the knowledge we have gleaned from the star diagram to propose an interface for a module encapsulating the `rooms` variable. The uses in the initialization function can be replaced with a reader and writer function, or the code for reading the portion of the input file describing the puzzle's rooms can be converted into a "LoadRooms" function. A "PrintRoomDescription" function can be extracted from the `DescribeRoom` function. We can then create "CanMoveDirTest", "CannotMoveDirTest", and "GoDirection" functions from the uses of the `dir` field. Figure V.2 highlights the nodes in the star diagram that represent the functions in an interface for a `rooms` abstract data type.

The `adventure` example shows us that understandable star diagrams can be created for the C programming language. The star diagram provides the information needed

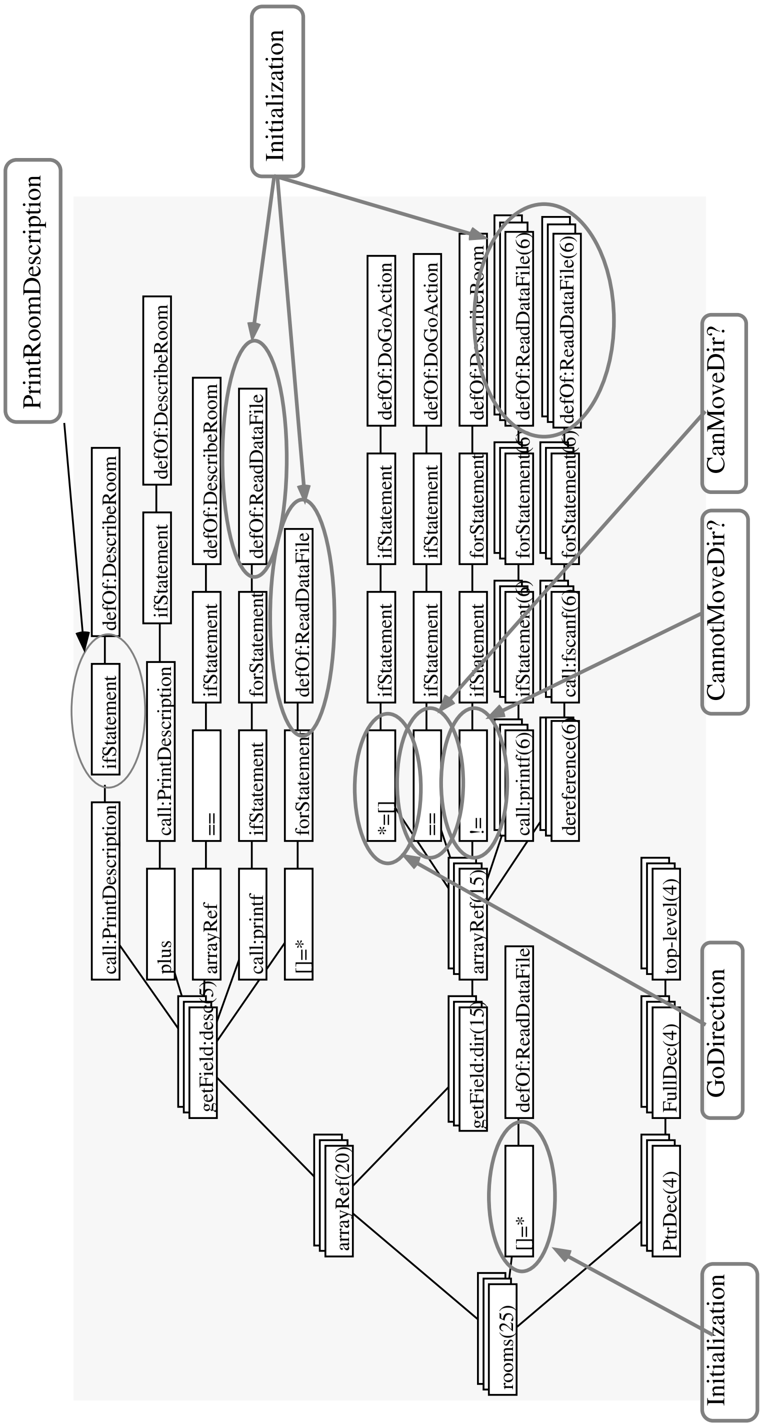


Figure V.2: Star diagram for the `rooms` array, with expressions representing abstract operations on the array labeled.

```
filen.c.cpp:83:extern Room *rooms;
filen.c.cpp:224: rooms = (Room *) malloc((numRooms+1)*sizeof(Room));
filen.c.cpp:226: fscanf(file,"%d %d %d %d %d %d",&(rooms[i].dir[0 ]),
filen.c.cpp:227: &(rooms[i].dir[1 ]),&(rooms[i].dir[2 ]), &(rooms[i].dir[3 ]),
filen.c.cpp:228: &(rooms[i].dir[4 ]),&(rooms[i].dir[5 ]));
filen.c.cpp:229: rooms[i].desc = GetAString(file,240);
filen.c.cpp:231: printf ("Room: %s  %d %d %d %d %d %d\n",rooms[i].desc,
filen.c.cpp:232: rooms[i].dir[0],rooms[i].dir[1 ], rooms[i].dir[2 ],
filen.c.cpp:233: rooms[i].dir[3 ],rooms[i].dir[4 ], rooms[i].dir[5 ]);
saadvn.c.cpp:79:extern Room *rooms;
saadvn.c.cpp:102:Room *rooms;
saadvn.c.cpp:194:   if ((rooms[currentRoom].desc)[0] == '*') {
saadvn.c.cpp:195:     PrintDescription(rooms[currentRoom].desc+1);
saadvn.c.cpp:198:    PrintDescription (rooms[currentRoom].desc);
saadvn.c.cpp:215: if (rooms[currentRoom].dir[i] != 0) {
saadvn.c.cpp:732:      if (rooms[currentRoom].dir[noun-1] == 0){
saadvn.c.cpp:745:        currentRoom = rooms[currentRoom].dir[noun-1];
saveFilen.c.cpp:69:extern Room *rooms;
```

Figure V.3: Output of `grep` when used to find occurrences of the `rooms` variable.

to plan encapsulations, and displays an overview of how the represented code behaves, as evidenced by our ability to identify the rough structure of the `rooms` variable from this view. The star diagram for `rooms` also compares favorably to other approaches to plan encapsulations in C programs. Although the diagram takes more space than the eighteen lines of `grep` output needed to display all uses of the variable (Figure V.3), the star diagram also groups similar expressions and allows us to quickly perceive a complete interface.

V.A.3 Language Features that Complicate Creating the Star Diagram

Although the above example shows that the star diagram can be used for C programs, features in C and in other languages complicate creation of the star diagram. Specifically, text-based macro languages that must be preprocessed before parsing hide macro abstractions from analysis tools that rely on parsing, and the language grammar can affect the amount of information provided by the star diagram.

Difficulties of Identifying Abstractions in Macros

Although C star diagrams appear to provide program details needed for encapsulation, not all abstractions are visible in the star diagram. Specifically, some programming styles encourage programmers to use macros to generate visible abstractions in the source code. Macro processors commonly used by programmers include `cpp`, the preprocessor used for the C programming language [Kernighan & Ritchie 78], and `m4` [Kernighan & Ritchie 86].

Macros generally are removed via a preprocessing step before compiling. As a result, any tool that relies on parsing source code in a language with preprocessor macros only sees expanded macros and not the uses of the macros. The star diagram, by relying on a parsed representation of the code, also cannot view macro uses. Although the macros may not hide a majority of the variable references, macros do represent important information about existing data abstractions in the code, and so the star diagram must highlight uses of macros as indications of logical operations.

For example, the source code for version 19.22 of the `emacs` editor [Stallman 93] is written in a highly portable style and runs on multiple platforms. As a result of portability constraints and style, the `emacs` source code frequently uses macros and conditional compilation to create multiple versions of the program from a single source tree, and to provide code abstractions that are implemented as inlined code, rather than function calls.

One of the variables in `emacs`, `minibuf_window`, contains a pointer to a window structure that occurs 57 times throughout the `emacs` source code, and is frequently acted upon by macros. When we examine the star diagram and the corresponding source code (Figure V.4), we occasionally cannot find the expressions represented. For example, in the star diagram for `minibuf_window`, the list of nodes

`minibuf_window` → `&` (bitwise AND) → `cast`

in the center of the star diagram represents the expansion of two consecutive macros uses. However, no code with these operations exists in the source code. The original code appears as:

```
WINDOW_FRAME(XWINDOW(minibuf_window))
```

(These macros strip type information for `emacs`'s Lisp interpreter from the `minibuf_window` pointer.)

Identifying macros for C star diagrams cannot be performed during or after

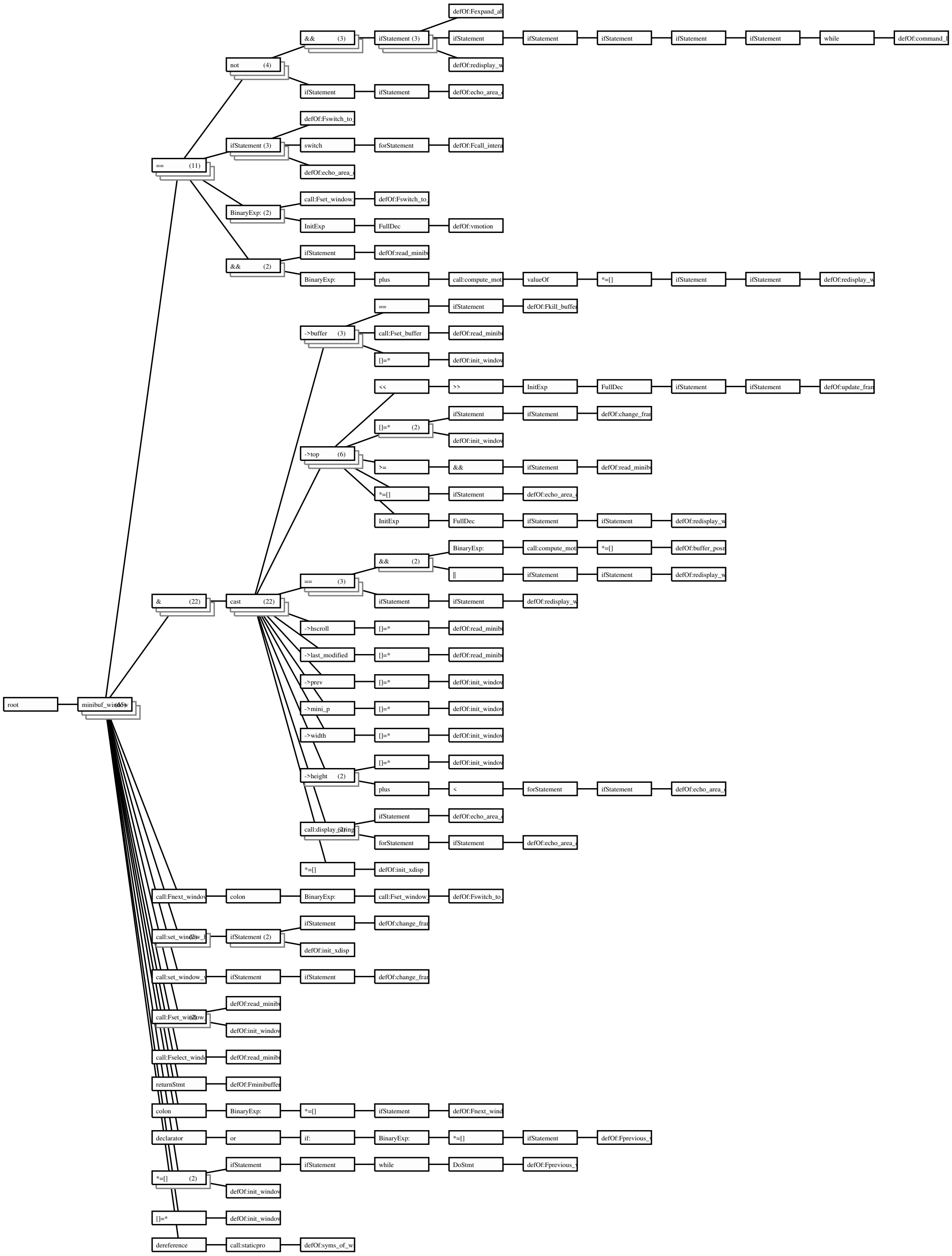


Figure V.4: Star diagram for the `minibuf_window` variable in GNU Emacs version 19.22. The variable occurs 57 times in a 76,000 line program.

parsing because the program may not be syntactically legal C code until after preprocessing. Preprocessing macros allows us to parse the source code and create the AST, but strips out one kind of abstraction that may be in a program. A better approach, suggested by Atkinson, Griswold, and McCurdy, uses an extended parser that attempts to process macros during parsing [Atkinson et al. 95]. If the code is a macro mimicking an inlined function, then the parser simply treats the macro as a function call. When creating the star diagram arm, the star diagram algorithm then can display macro uses as function calls and macros using a variable as a function definition. When a macro cannot be parsed, however, it would need to be inlined, with some indication in the AST that the code does not occur literally in the original source code.

Thus, although macros can complicate creation of the star diagram, parsers that identify instances of macros can ensure that macro information is available in the star diagram for presentation to the user.

Language Grammars

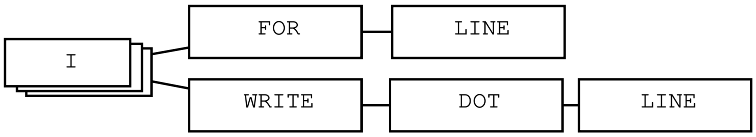
The grammar of a programming language also significantly affects creation of the star diagram. In particular, the structure of a language's grammar may not allow the star diagram algorithm to infer data and control dependences from nesting relationships. The AST form of a statement also may not match the tool user's understanding of the code, and thus reduce the ability of the star diagram to convey the behavior of the code to the star diagram tool user.

As described in Chapter III, the star diagram algorithm uses the AST of a Scheme program to infer data and control dependences from statement nesting. Not all languages support this. For example, MUMPS is a database language similar to BASIC that is frequently used in health-care systems. The language was first created in the 1960's, but has been extended over time to support larger databases. MUMPS is a line-based, interpreted language. Statements are parsed in a line-by-line fashion.

Because code is parsed on a line-by-line basis, the AST does not directly connect statements normally joined in the C or Scheme star diagrams. For example, when an

```
FOR I=0:1:10 DO
.WRITE I
```

(a) MUMPS code



(b) Star diagram for code

Figure V.5: Fragment of MUMPS code performing a loop, and a naive star diagram for it. The `FOR` command performs the body of the loop ten times, incrementing the value of `I` from 0 to 10. The `DO` command at the end of the line normally performs a procedure call, but with no parameter, it treats succeeding lines with a dot prefix as part of the same scope, and thus part of the `FOR` loop. As a result, the `FOR` loop extends across a line boundary. Because parsers for MUMPS would normally parse the code line-by-line, the AST would not identify that the use of `I` on the second line appears inside a loop.

`for` loop extends across separate lines, portions of the `for` on the next line occur in a syntactically independent unit. As a result, finding the successor of a given node in the star diagram cannot be done simply by moving up the AST, but sometimes requires examining the code (and, in effect, identifying control dependencies between operations) to identify the matching statement. Figure V.5 shows a example of such code.

Interpreted languages like MUMPS are not the only languages where the AST does not provide enough information on “related” statements. A star diagram for an assembly language would receive no information from any source code representation of the program, except identifying which statements are near each other. To generate a star diagram for an assembly language program, control flow analysis is needed to identify statements depending on a conditional operation, and dataflow analysis would be needed to see where a register is next used.

Figure V.6 shows a possible star diagram created from uses of the `MAJOR` variable in the pseudo-assembly language shown. Note that the successor function for an assembly language star diagram must follow must follow register assignments to the uses of that register in order to identify cohesive calculations. The “LOOP” node represents a control construct like the `For` or `Do` loops in C or Scheme, except the construct must be inferred from the branching behavior of the code. In this case, seeing that the branch causes a previous calculation to be repeated might suggest that the resulting value depends on the

```
;;; Sample pseudo-assembly code
;;; This code contains a pointer to a major data structure, MAJOR,
;;; arranged as a set of 10 byte records.  The user wants to access
;;; a given record number (passed in register B), and have it returned
;;; in a location known as TMP_PTR.
;;; Side effects: destroys R1,B

GETPTR
        LD      R1,(MAJOR)  ;; put MAJOR's contents into register 1
GETPTR10
        ADD     R1,10,R1      ;; add r1 and 10, put result in r1
        DEC     B             ;; decrement the record counter
        BRANCH  zero,GETPTR10 ;; and loop until we're pointing to the
                              ;; record we want
        LD      (TMP_PTR),R1  ;; Save ptr value into TMP_PTR location.
        RET
...
;;; Trash the first record of MAJOR by incrementing the pointer in front
;;; of it.
REMOVERECORD
        LD      R1,(MAJOR)
        ADD     R1,10,R1
        LD      (MAJOR),R1
        RET
```

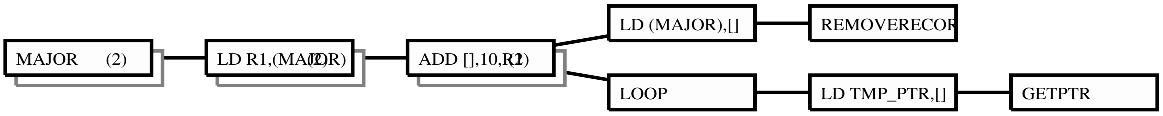
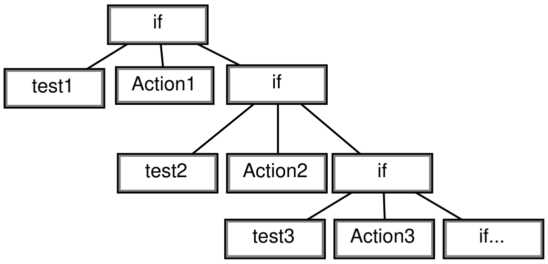


Figure V.6: An example of how a star diagram for an assembly language program might appear. In this case, the star diagram is for the variable `MAJOR` in the fragment of assembly code provided. Note that because we cannot infer dataflow from the nesting of expressions, uses of a given assignment to a register must be traced.

loop. Alternatively, if we had full dataflow information, we could identify whether the loop truly affects the value of the variable. As seen, the star diagram roughly describes the computation performed, and would combine other, similar computations that occur on the variable `MAJOR` throughout the code.

Although lack of syntactic nesting can complicate creation of the star diagram, nesting can also be misleading. For example, in C, chains of `if-then-else` clauses may appear as a linear set of statements, but are actually nested syntactically. A variable use occurring in the last clause creates a star arm with a long sequence of `if` nodes representing




```
if (test)
        Action1();
    else if (test2)
        Action2();
    else if (test3)
        Action3(*line-storage*);
    else if ...
```

Figure V.7: The nesting of certain code structures (in this case, in the C language) sometimes creates large AST structures and long star diagram arms.

---

all the previous if clauses. Figure V.7 shows the code and AST for a chain of nested if statements. The code parses into a form where the `if` node has, as children, the `test`, `then` and `else` clauses. The `else` clause represents the next `if` statement, whose `else` statement contains the next `if` statement, and so on. Thus, the AST rooted at the first `if` contains a long, narrow tree representing all the other `if` statements.

Although, syntactically, the first `if` statement may enclose all other `if` statements, the programmer might consider each statement to be one member of a set of independent comparisons (as the `cond` operation in Lisp appears, or as a programmer might interpret the tests in a `switch` statement in C), and so creating a long star arm where the last statement depends on the first may not match the programmer's interpretation of the statement. In this case, the parent relation in the AST does not always logically represent the "consumer" or "controller" of the result of an expression, even when the control flow for the statement clearly indicates the last test depends on the first. As a result, the **Successor** or **SkipNodeTest** functions for C star diagram algorithm may can be adjusted to display only one of these `if` statements in the star diagram, and ignore the others.

V.A.4 Summary

Star diagrams can be used for languages other than Scheme. The creation of star diagrams for C shows that the algorithm provided in Chapter III easily accommodates new languages. Adjusting the star diagram algorithm to handle both C star diagrams that support variable encapsulation (like the Scheme star diagrams), and the new type-oriented star diagrams require changes only to the language-dependent portions of the star diagram algorithm. The simplicity of the modifications for supporting a new language and the speed of creating the star diagrams show the feasibility of moving the star diagram concept to other languages. Our C example shows that we can understand code and plan an encapsulation from the star diagram view. Finally, we find language features of each language can introduce new difficulties into creating the star diagram. Not all languages allow data and control dependences to be inferred from statement nesting, and so creating star diagrams for these languages may require additional analysis.

V.B Scalability of the Star Diagram Concept

To plan the encapsulation of a variable in large C and MUMPS programs, the most frequently accessed parts of a star diagram need to be viewable as a single understandable unit. If the star diagram is too large, the tool user cannot examine the entire star diagram at once, and instead must memorize relevant details and scroll through the picture to plan a global change. Without an overall view, the tool user may also make local, non-optimal restructuring decisions that result in a poor choice of interface functions. Consequently, scalability of the star diagram concept is essential.

To identify how well the concept of the star diagram scales, we compare the sizes of the star diagrams for small and large programs. Our measurements show that for larger star diagrams, both the vertical and horizontal size of star diagrams may be too large to easily present to the star diagram tool user. Some scalability difficulties are easy to solve, such as limiting the length of star diagram arms by eliminating uninteresting parts of the star diagram. However, the vertical size of the star diagram representing a variable used thousands of times may always be much too large to view on a single computer screen. Adaptations of the star diagram concept may still allow restructuring via a single, concise view.

We will first discuss our logical expectations about the size of the star diagram based on the star diagram algorithm (Section V.B.1), list the programs we examined and metrics we collected (Section V.B.2), and describe how the vertical and horizontal size of the star diagram[1] increases as program size and number of variable uses increase (Sections V.B.3 and V.B.4).

V.B.1 Predicted Behavior

Before beginning to examine actual star diagrams, we can identify the likely appearance of star diagrams by considering the behavior of the star diagram algorithm and the form of the AST.

We can expect that the length of star diagram arms (the horizontal size of the star) will remain short. Because we generate the star diagram by "walking up" the AST, the star diagram arm lengths will be proportional to the height of subtrees within the AST. If the tree is balanced, the height is the logarithm of the number of nodes, and so the amount of code within a subtree of the AST grows roughly exponentially with each step. Because each subtree of the AST also only represents the expressions in a given function, not in the entire program, the total size of any AST for a function remains small, and thus limits the height of the AST. Thus, we can expect the horizontal size of the star diagram to increase with the average height of AST subtrees, which will grow logarithmically with the average size of the functions in the program. More importantly, the length of star diagram arms will be independent of the size of the program. Thus, we expect the horizontal size of the star diagram to be a minor concern when analyzing large programs.

The width of the star diagram (the vertical size of the star diagram) may be less

[1]In our discussion of star diagrams, the meaning of "length" of the star diagram graph and "width" of the star diagram graph correspond to the definition of these terms for graphs. However, these terms conflict with the idea of the "width" of the picture or page, and so we will generally use "vertical size" to refer to the width of the graph, and thus the height of the picture. We will use the term "horizontal size" to refer to the height of the graph, and thus the width of the picture.

scalable. Logically, the number of arms in the star diagram depends on the number of occurrences of the variable. At the root, all arms are represented by a single node; moving towards the right, the star diagram arms diverge. In the best case, variable uses occur in similar contexts so that the star diagram arms remain merged until near the function nodes. In the worst case, however, all the star diagram arms will occur in completely different circumstances, and diverge immediately. Looking at the star diagram from left to right, we would expect the star diagram to change from every expression represented by a single node at the root to every expression represented by a separate node on the right hand side of the star diagram. Such behavior would cause the star diagram's vertical size to be equal to the number of relevant variable uses in the program, and would indicate that the star diagram will grow directly with the number of references to a given variable. We can assume, however, that although the number of uses of a given variable can become extremely large, each data structure can only be used in a limited number of ways. We expect that by overlapping similar star diagram arms and stacking nodes, we can reduce the vertical size of the star diagram enough so that a star diagram user can quickly perceive the overall behavior of a given variable.

We can test these hypotheses about the vertical and horizontal size of star diagrams by creating star diagrams for real programs.

V.B.2 Measuring the Size of Star Diagrams

In order to test whether our hypothesis about the size of star diagrams holds true, we need to examine real star diagrams. We created star diagrams and measured their sizes for a range of programs, including all the star diagram examples described in previous chapters. We examined star diagrams for frequently and infrequently used variables. The star diagrams include Scheme, C, and MUMPS programs, and represent both variable- and type-oriented star diagrams.

The programs we analyzed include all examples given in previous chapters. We list the programs in Table V.1. For Scheme, we examine the star diagram in Figure II.6, the largest star diagram in the `KWIC` example. For the `adventure` program (described in Sec-

program	variable	size (pages) rows	columns	nodes	nodes unmerged
KWIC	`*line-storage*`	1	1	37	55
Adventure Interpreter	`currentRoom`	1	1	77	176
	`items`	2	1	126	493
	`rooms`	1	1	38	151
gnuEmacs	`minibuf_window`	2	2	139	279
	`current_buffer`	26	3	3,633	16,410
	`struct buffer`	29	3	3,741	17,893
CHCS (Mumps)	`DIC`	20	2	2,440	281,632
SunOS (UNIX)	`struct mbuf`	11	2	1,218	6,505

Table V.2: Measurements of overall size of the star diagrams examined.

tion V.A.2), we examine the integer variable `currentRoom`, the infrequently used array `rooms` described in the example, and the frequently used array `items`. We also create star diagrams for the `emacs` editor (briefly described in Section V.A.3). For `emacs`, we focus on the infrequently used variable `minibuf_window` and the widespread `current_buffer` variable. We also create a type-oriented star diagram for all the pointers to `buffer` structures. To compare the star diagrams for C and MUMPS programs, we examine the `DIC` variable from portions of the Comprehensive Health Care System (CHCS) source code, a half-million line hospital management system (described in Section V.A.3). Finally, to examine a large body of commercial C code, we create the star diagram to examine all variables of type `struct mbuf` in the SunOS networking code (originally described in Section I.A.1).

Table V.1 also lists the variables examined in each program, the number of references to the variable of interest throughout the program (or, in the type-oriented star diagram examples, the number of uses of variables of the given type). To compare the size of the star diagram to the size of the information presented by current methods for planning encapsulations, we also note the size of the `grep` output that would list all uses. The size of grep output does not always equal the number of references to the variable. For example, in larger programs, `grep` sometimes matches strings similar to the variable name that could

program	variable	distinct arms	length max.	length average	length median
KWIC	`*line-storage*`	13	9	5.5	5
Adventure Interpreter	`currentRoom`	24	7	5.1	5
	`items`	39	9	6.6	7
	`rooms`	12	7	5.7	6
gnuEmacs	`minibuf_window`	46	11	5.1	5
	`current_buffer`	932	27	10.6	9
	`struct buffer`	1029	25	9.6	8
CHCS	`DIC`	698	18	7.5	7
SunOS	`struct mbuf`	393	13	4.2	4

Table V.3: Measurements of size of star diagrams in number of star diagram arms (paths through the graph) and length of arms.

Program	variable	Level of graph: 1	2	3	4	5	6	7	8	9
KWIC	*line-storage*	5	6	7	5	6	4	1	1	1
Adventure Interpreter	currentRoom	1	9	16	17	15	11	7		
	items	1	3	6	19	29	28	20	14	5
	rooms	1	3	3	7	10	8	5		
gnuEmacs	minibuf_window	8	20	19	23	27	16	11	7	33
	current_buffer	4	63	182	243	304	331	300	263	224
	struct buffer	1	71	218	288	366	393	337	281	221
CHCS	DIC	2	25	112	274	423	479	413	290	182
SunOS	struct mbuf	166	265	290	228	132	67	35	17	9

Table V.4: Measurements of width of star diagrams, in number of nodes at each level of the graph.

Program	variable	size of average stack	
		all nodes	stacked nodes
KWIC	`*line-storage*`	1.5	5.5
Adventure Interpreter	`currentRoom`	2.3	4.3
	`items`	3.3	4.7
	`rooms`	3.6	7.2
gnuEmacs	`minibuf_window`	2.1	7.3
	`current_buffer`	3.6	9.4
	`struct buffer`	3.2	7.4
CHCS	`DIC`	96.3	179.9
SunOS (UNIX)	`struct mbuf`	3.8	4.8

Table V.5: Average stacking occurring on star diagram nodes.

Program	variable	number of nodes	
		stacked nodes	unstacked nodes
KWIC	`*line-storage*`	4	33
Adventure Interpreter	`currentRoom`	30	47
	`items`	83	87
	`rooms`	16	32
gnuEmacs	`minibuf_window`	21	144
	`current_buffer`	1606	3668
	`struct buffer`	1889	3774
CHCS	`DIC`	1300	1140
SunOS (UNIX)	`struct mbuf`	1570	559

Table V.6: Amount of stacking occurring on star diagram nodes.

not be easily filtered out. If multiple occurrences to the variable occur on the same line, `grep` shows the line once. `grep` also cannot not identify all uses of all variables of a given type with a single query.

Table V.2 describes the overall size of the star diagram, both in the pages required to draw the star diagram and number of nodes in the star diagram. To judge the amount of compression accomplished by stacking nodes, we measure both the total number of nodes in the diagram, and the number of "unmerged nodes" if we performed no stacking or matching of common expressions.

The "number of pages" figures describe the size of a printed version of the graph. For large graphs that cannot be displayed on a single sheet, we print multi-page graphs that can be taped together and displayed on a wall. Pages are printed in portrait mode. Our choice of font and spacing allows us to fit about 36 star arms vertically on a page, and 10 nodes of a star diagram arm horizontally. Although a programmer may not be able to directly perceive all the information in a multi-page star diagram, having the star diagram available as a single sheet does permit gaining a global view of the program, especially when the programmer can highlight the printed copy to mark interesting code fragments. The greatest limitation of the paper version of star diagrams is they do not easily support mapping from a node on the paper to the associated source code. Figure V.8 shows the rough appearance of the multi-page star diagrams.

V.B.3 Horizontal Scalability of Star Diagram Arms

Using the metrics we have gathered, we can examine the star diagrams shown to identify the scalability of the star diagram concept. The first constraint on the star diagram's size is the length of the arms of the star diagram. The horizontal size of the star diagram must be small enough that the tool user can identify star diagram nodes of interest without scrolling.

When we examine the length of star diagram arms in the sample stars (Table V.3), we confirm our expectation that the horizontal size of the star diagram remains small. The horizontal size of star diagram appears independent of program size, The average length of

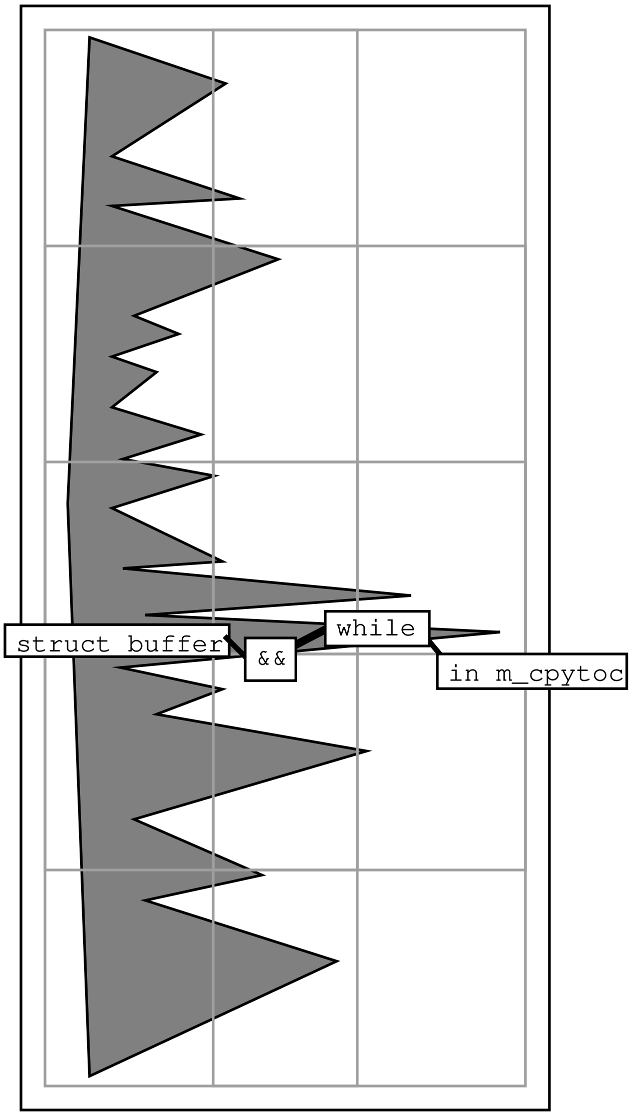


Figure V.8: Conceptual image of a multi-page star diagram. The gray lines represent individual pages boundaries. The star diagram shape (in dark gray) approximates the actual appearance of large star diagrams. One star diagram arm is included to indicate orientation of the drawing. This image highlights the general appearance of multi-page star diagrams. The star diagram expands to its full vertical size within the first two levels of the graph. Most star diagram arms never reach the right side of the first page. The few long star arms represent either complex expressions, such as particularly complex macros, or are caused by nested `if` statements.

arms in the star diagram is always under ten nodes, and usually under seven. Because our star diagram printouts usually fit ten nodes on an 8 1/2" wide paper, and the Scheme star diagram generator can easily show twelve to fifteen nodes on a 19" computer screen, we can assume that a significant fraction of a star diagram graph can fit horizontally on paper or screen.

Although the average measurements suggest that the star diagram's horizontal component will usually fit on a page or screen, the maximum lengths of star diagram arms (shown in Table V.3) always appear to require two or three paper pages to display. When the length of the star diagram arms becomes larger than a screen width, the star diagram tool user can no longer see the right hand side of the star diagram, and will be forced to either scroll to see the necessary information, or plan the interface with imperfect information.

Some of these long star diagram arms are caused by language structures that create uninteresting and long star diagram arms (as described in Section V.A), and can be solved by removing the uninteresting nodes.

Most other long star arms are simply caused by the complexity of expressions and functions in the code being examined. We can avoid such scalability problems by displaying only the details of the star diagram immediately useful to the programmer. For example, to plan an encapsulation, the tool user needs information on both the far left and far right sides of the star diagram. The left hand nodes represent the common, small expressions in the code that are likely candidates for new functions, while the function definition nodes at the far right of the diagram describe the context of the uses, as we saw in the `adventure` example. In fact, we will see in our observations of programmers (Chapter VI) that users of the star diagram tend to begin at the left of the diagram looking for appropriate abstractions, and examine the right hand side of the star diagram to identify the context of uses. They then choose functions represented by nodes located towards the left hand side of the star diagram.

To avoid cases where the tool user may miss information because of the length of star diagrams, we can leverage the observed browsing style and highlight the information most important to the encapsulation (the stacked nodes and scope information) while eliding the middle of the diagram. Arms extending beyond the window's right border can be elided so as to include the needed function definition nodes at the edge, and the hidden nodes could be displayed when the programmer scrolls to the right.

These measurements of the star diagram's horizontal size suggest that star diagrams may usually be too wide to fit on a single page or screen. However, we also see that star diagrams will not exceed a few pages or screens in width except in less interesting cases such as when the AST displays irrelevant, large, nested expressions. The simple scaling techniques presented here, such as eliding the central, and less important, sections of the star diagram can allow all star diagrams to fit across a screen or page, and permit the programmer to view the most important parts of the star diagram in a single view.

V.B.4 Vertical Scalability of the Star Diagram

Although a star diagram may fit across a page, the vertical component of a star diagram proves less manageable. This second constraint on the star diagram is mainly dependent on the number of arms in the star diagram. Although we hoped that the grouping of nodes in the star diagram would limit the number of distinct arms in the star diagram and keep the size of the star diagram manageable, we find that star diagrams for frequently used variables in large programs still require multiple pages to display. We find that although the star diagrams will not easily fit in a single, small, and understandable unit, we can apply a number of scaling techniques to minimize the information the star diagram user must examine.

We can first examine the vertical size of the star diagrams examined, and compare the size of the star diagrams to the current methods used to plan encapsulation. Table V.2 lists the size of the resulting star diagrams for each variable, and Table V.1 lists the size of the `grep` output. The star diagrams require roughly the same space as the printed `grep` expressions (assuming 56 lines of grep output per page), even though the star diagrams provide additional information not found in `grep`. For variables occurring less than a hundred times, we find that the star diagram can fit on a single page. (Although Table V.2 shows that `items` and `minibuf_window` require multiple pages, they only partially exceed one page, and can fit on a single page via slight reduction.) The larger star diagrams vary. For the `emacs` examples and for the `struct mbuf` example, the star diagrams require roughly the same space as `grep`—for example, displaying the uses of current_buffer requires 24 pages for the `grep` output and 26 pages for the star diagram. The `DIC` star diagram, with several hundred pages of `grep` output, fits on twenty pages, partially due to significant stacking of nodes.

Even at twenty pages, the star diagrams are much larger than we would hope because such a large star diagram will not be viewable as a single unit on a computer screen. It could be argued that our scale problems are the result of trying to encapsulate excessively complex data structures. For example, in `emacs`, the `current_buffer` variable points to a data structure with thirty fields; in this case, even a simple reader and writer function for each field would require sixty functions. However, the size of such an interface is comparable with the number of functions in Text widgets for GUIs. For example, the Text class in NeXTStep user interface requires 41 instance variables and sixty methods directly concerned with text processing and manipulation [NeXT Computer, Inc. 90]. Although `current_buffer` may not be a typical variable to encapsulate, a programmer certainly could wish to encapsulate it with a restructuring tool.

Assuming that these star diagrams represent reasonable variables to encapsulate, we find that the problems exhibited by the size of the star diagrams fall into two categories: manipulability and focus.

First, these star diagrams may be difficult to manipulate and examine. Stretching the star diagram for `current_buffer` over 26 screens will introduce problems for the programmer trying to gain a global view of the system. Visualization techniques such as including a small overall view of the visualization (on the scale of Figure V.8) can reveal where in the large star diagram the current view is located, and can help the programmer move through the star diagram efficiently. However, displaying star diagrams on a computer screen does mean that the programmer can only look at a small number of star diagram nodes at one time. The programmer must search multiple times through the code to identify related details.

For such large star diagrams, however, the paper approach is more usable in its current state simply because the programmer can see more detail at once. The largest star diagram example, `struct buffer`, would fit on ten sheets of drafting paper. Equivalent numbers of drawings in other engineering disciplines would be considered understandable, especially because the paper designs also allow other methods of browsing, such as placing two sheets next to each other, folding, and highlighting interesting portions of drawings. To use paper star diagrams in a real setting, some method of mapping from a star diagram to the relevant code (such as indexing nodes) in a star diagram tool could allow the programmer to identify an interesting node in the star diagram, then examine the relevant lines of code.

If we want to examine these star diagrams on computer screens (or if we wish to make the paper star diagrams less awkward), we must reduce the vertical size of the star diagrams by focusing the user's attention on specific portions of the star diagram graph. Because the primary contribution to the size of star diagrams is the number of diverging arms, we will need to reduce the number of arms in the star diagram in order to remove the scale problems we have seen,

However, most simple approaches will not help us examine the star diagram. For example, showing only a fixed number of levels of the star diagram graph might limit the number of star diagram arms seen and thus limit the vertical size of the star diagram. However, because the star diagram expands so quickly, such an approach cannot both reduce the number of star diagram arms and still provide meaningful context to the tool user. Table V.4 shows how the star diagram quickly expands to its full height within two to three levels of the graph, and usually with one half to two thirds of the star arms passing through any given level of the graph. Thus, to limit the vertical size of the star diagram, the star diagram would need to be trimmed so close to the root that only one or two operations near each variable would be visible in the star diagram.

We could also restructure large star diagrams in sections. When encapsulating structures with a large number of fields (such as the `current-buffer` structure in `emacs`), the star diagram user may be able to encapsulate individual fields of the data structure separately if the sections are sufficiently independent. For example, it might be feasible to first create functions for manipulating individual fields, then create functions for manipulating the entire data structure.

Although this approach could allow a programmer to make ill-advised choices, careful identification of the independent structures could allow programmers to manipulate portions of a star diagram in isolation. For example, the tool user could partition a star diagram according to the category of operation. For example, in the star diagram for `struct mbuf` example, we partitioned the star diagram into sections for (a) field accesses, (b) manipulations of the `mbuf` structure itself, and (c) function calls passing `mbuf` structures. By assuming that field access operations are probably cohesive with other field access operations, the tool user can examine each section independently.

Another alternative is to try to limit the amount of information displayed to the tool user by further clustering related nodes. Condensing the existing star diagrams for an overall view can be performed in three ways:

- **Assume only stacked nodes are interesting.** First, the nodes that we are most likely to want to turn into functions are those that represent common expressions. These appear as stacks of nodes in the star diagram. The programmer could choose to only examine these stacked nodes. As Table V.5 and Table V.6 shows, the stacked nodes usually represent about a third of the total nodes, reducing the size of the largest star diagrams significantly, depending on the number of arms that contain stacks. We can expect a significant reduction in the size of the star diagram, but perhaps not enough to permit the star diagram to appear on a single screen. However, this solution could cause the programmer to overlook important computations that occur only once in the program that represent major functions in the interface, such as the initialization functions in the `rooms` encapsulation.

- **Loosen the stacking criteria to avoid diverging star arms.** Second, we could change the successor function of the star diagram algorithm so it skips over certain uninteresting nodes, and thus reduces the amount of divergence in the star diagram. We could hide the less interesting scope nodes such as uninformative `for` and `if` nodes representing large blocks of code.

For example, Table V.6 suggests that in the case of `emacs` that stacking may be low because many nodes represent a single unique node. Because `emacs`'s star diagrams had the longest arm lengths, we can guess that the `emacs` source code contains more scope nodes, and as a result, the long arms contain many scope nodes towards the right that are not matching. Because these scope nodes represent large `for` loops or `if` clauses occupying much of the function, changing the star diagram to ignore scope nodes could allow star diagram arms to remain merged for their whole length.

We could also truncate longer star arms at some given length. (Chen implements this in his restructuring tool.) By reducing the horizontal size of the star diagram, we can remove some branching nodes, and thus reduce the vertical size of the star diagram.

- **Identify and group concepts, not nodes.** Finally, we could use automatic concept assignment tools to identify expressions likely to represent functions, and then either sort arms to highlight the likely functions, or hide the expressions not likely to form functions.

  With appropriate analysis, the restructuring tool could display low-level concepts, rather than expressions. Star diagram nodes could represent multiple uses of a variable that match a particular common operation as a single star diagram node, thus reducing the number of star diagram arms and the amount of information the programmer must examine.

  For example, the code:

  ```
  ptr = ptr->next;
  ```

  uses the `ptr` variable twice, and so creates two arms in the star diagram. By identifying the code as a single node labeled "AdvancePtr" in the star diagram, we eliminate one of the uses from the star diagram, and hopefully reduce the size of the star diagram. By automatically identifying expressions that enclose multiple uses of the variable, more information is made available to the programmer about the kinds of operations being performed on the code, and fewer star diagram arms need to be displayed.

The techniques used could be as complex as recognizing plans, schemas, or cliches [Rich & Waters 88, Quilici 93] or as simple as identifying common coding styles in the language, such as moving a pointer down a linked list of nodes. More promising approaches may allow the user to describe frequently used domain- and organization-specific coding styles and code fragments that represent single concepts. However, in order to reduce the size of the star diagram significantly, the matching algorithms need to identify a significant number of concepts, each representing multiple uses of the variable or type, to reduce the size of the star diagram. The concepts recognized would also need to be at a slightly lower level than the functions being created in order to allow the programmer to identify and choose conceptual operations.

These solutions suggest that larger star diagrams can be examined with some scrolling, or by examining paper copies to plan the restructuring, then manipulating the actual star diagram. Reducing the vertical size is the main impediment to scalability for the star diagram.

V.B.5 Discussion

Examining the star diagrams for C and MUMPS programs shows us that scalability is a significant challenge. First, we have learned that star arms tend to be short. Although a star diagram may not always not fit horizontally on a single screen or page, elision techniques can ensure that the most important parts of a star diagram arms are visible. Logically, the most important parts of the star diagram would be the left half, where the common expressions likely to be turned into functions are represented, and the right side where function context is listed.

Second, the star diagram's vertical size may be an impediment to easily examining the entire star diagram. Although the height of the star diagram may appear to be proportional to the number of references to the data structure being examined, larger programs do exhibit the property that the variable is only used in a given number of contexts, thus limiting the number of contexts the star diagram must show. A number of approaches can be applied to further reduce the size of the star diagram and improve the visibility of essential information: ignore singleton expressions, use other matching methods to identify and display expressions that are likely targets for restructuring, loosen the matching function's discrimination, sort elements by likelihood of being a function, or attack large star diagrams in sections. More intelligent methods for determining what to present may be necessary to reduce the size of the star diagram to manageable levels.

V.C Effect of the Unimplemented Aspects of the Algorithm on Star Diagram Size

The star diagrams presented in this section, as mentioned in the introduction, are simplified forms of the star diagrams described in Chapter III. These simplifications fall into two categories: eliminating function nodes and ignoring edges representing dataflow through local variables. Although these simplifications affect the exact measurements of star diagrams, they do not change our conclusions about star diagram size. We will address both and discuss how star diagram appearance might change if these features were included in the C and MUMPS star diagrams.

V.C.1 Eliminating Function Nodes

The star diagrams measured in this chapter are simplified by eliminating function nodes from the right hand side of the star diagram. This change was caused by the lack of a directed graph layout algorithm in our current environment. When we laid out star diagrams using trees, we found that the tree layout approach caused significant problems for large programs. CHCS was particularly vulnerable to scale problems because of the sheer number of references and because specific operations were not localized to a small set of functions.

Using a tree layout for an extremely large star diagram causes the size of a star diagram to explode primarily because of excessive function nodes. If a common expression occurs in two hundred functions, then the tree layout causes the chain of nodes to diverge into two hundred separate arms (or eight pages of a printed star diagram). When a star diagram contains a large number of common expressions, this fanout problem can make the star diagram almost impossible to generate. For example, when creating the star diagrams for the MUMPS CHCS program, the version of the `DIC` star diagram with function nodes created a star diagram at least *281 pages high*; we stopped generating the star diagram at that point. By contrast, the star diagram without function nodes can be drawn in twenty pages.

If we had drawn the C and MUMPS star diagrams as directed graphs, we might have found that, as in the Scheme star diagrams, the function nodes help the tool user identify variable uses in the same context. However, we could also imagine that for large star diagrams representing variables used in thousands of functions, the right hand side of the star diagram could quickly become a rat's nest of crossing lines as the algorithm tried to connect star arms with the containing function. To improve the graph's appearance in such circumstances, the star diagram could simply draw one function node for every star diagram arm, as the C star diagrams currently do, thus trading off graph complexity for contextual information. Another approach for displaying context that does not require direct connections would be to allow the programmer to select a star diagram node, then either highlight star arms in the same context or create a smaller star diagram containing only the star diagram arms in that scope. Both solutions force the programmer to explicitly decide to ask "which other uses of the variable occur in this scope?", rather than automatically showing context and letting the programmer identify functions that seem to be connected to many uses.

V.C.2 Ignoring Star Arms Representing Dataflow though Local Variables

Our second simplification of the star diagram algorithm for this chapter is to eliminate star diagram edges caused by dataflow through local variables. Because of the implementation cost of def-use analysis of the variables within the programs, dataflow edges could not be introduced into our current star diagrams. This potentially affects the star diagrams in this chapter in two ways. First, the diagrams might be generated artificially fast. However, def-use analysis, especially for local, scalar temporary variables, is relatively cheap to perform, and so should not lengthen star diagram generation time.

Second, the star diagrams could expand in size. However, we can see in Table V.3 that the Scheme star diagram's arms are not significantly larger than the star diagram arms for the C and MUMPS star diagrams, even though the Scheme star diagram arms are longer due to the dataflow edges. Adding data flow edges may add additional paths to the star diagram when a temporary variable is used in multiple contexts, and thus increase the vertical size of star diagrams. Because assignments to temporaries occur infrequently in the C star diagrams, we can expect the vertical size of the star diagram should not increase significantly due to the addition of dataflow edges.

Thus, although the star diagrams shown do not fully implement the star diagram concept as presented in Chapters II and III, we can expect that star diagrams including dataflow and function nodes do not have to be significantly larger than the star diagrams presented.

V.D Summary

This study of large star diagrams for programs up to a half million lines in size, in languages other than Scheme, suggests the following attributes of the star diagram:

- Star diagrams can be generated for languages other than Scheme.
- Language grammar and AST structure affect the star diagram algorithm. However, many of the difficulties can be eliminated by adjusting the parameterized functions to the star diagram algorithm. In the extreme case of assembly language, the AST provides no information that the star diagram can use, and as a result depends entirely on def-use analysis and control flow analysis to identify related expressions. Other languages require a mix of using the AST for identifying containing expressions with using data and control flow to identify related operations.

- By building star diagrams from a parsed form of the languages, language features such as preprocessor macros can complicate creation and presentation of the star diagram.

- Although the star diagrams are too large for examining at one time on a computer screen, they do appear to be usable, and additional work on limiting the information shown to that most important to the programmer should reduce star diagrams to a manageable size.

Chapter VI

Observations of Programmers Restructuring

While we can argue that restructuring is a difficult and tedious process that requires a global understanding of the program, we have little hard data on exactly how programmers restructure programs. In the case of the star diagram, we have little evidence to decide empirically that it addresses the problem of encapsulation.

To investigate whether how the star diagram changes the restructuring task as performed with standard programming tools, we examined how programmers perform restructuring with different toolsets. We employed systematic observational techniques on pairs of programmers using either a prototype restructuring tool with the text-oriented interface, the star diagram visualization, or traditional UNIX editing and searching tools.

The basic question we asked was: "how do programmers use the capabilities of each set of tools to guide their progress in the restructuring task of data encapsulation?" In short, we discovered:

- Programmers were able to successfully restructure using any of the tools.
- We created a framework of six tasks, abstracted from the programmers' actions, that describes how programmers perform restructuring. We can use this model to evaluate and design this tool and future restructuring tools.

- We observed that “bookkeeping”—keeping track of progress in the overall restructuring task and in specific modifications—is a crucial activity during restructuring. To facilitate bookkeeping during the task, each group exploited structure implicit in the tools and program representation to maintain state. Their particular choice of structure used exchanges the possibility of some types of errors for others.

- Limited bookkeeping support led to more planning of the restructuring before beginning modifications. However, the plans suffer because of how the programmers must accommodate the needs of bookkeeping.

- The star diagram provided support for bookkeeping and planning not found in the other tools. However, its support for manipulation and bookkeeping encouraged an exploratory style that eschewed long-range planning. The star diagram’s bookkeeping also supported low-cost backtracking to recover from design oversights. This model may scale better to larger programs.

We first describe the study methodology. We justify our decision to use *constructive interaction* techniques to observe teams performing restructuring, and to systematically analyze the resulting videotapes and logs. We also describe the experimental setup and reasons for those decisions (Section VI.A). Second, we provide an overall framework describing our observations, propose a model of programmer behavior and describe each team’s overall behavior (Section VI.B) Third, we describe in detail how each team used support provided in the tool environment to perform the change without introducing errors. Fourth, we discuss how the choice of strategy for ordering tasks in the transformation significantly affected the types of errors made (Section VI.C). Finally, we discuss why our observations should apply to programmers modifying larger programs written in common programming languages such as C.

VI.A Study Method

VI.A.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 92], 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. Finally, because little is known about these tools, it is not possible to isolate and test a few experimental variables. Also, Schneiderman and others have noted that understanding the context of usage is crucial to understanding how software tools are used [Schneiderman & Carroll 88], and so unless we understand the overall context of restructuring and encapsulation tasks, we may incorrectly design a restructuring tool.

As a result, we chose a class of exploratory methods called systematic observational techniques. Systematic observational techniques involve a detailed analysis of a small number of subjects, and thus produce qualitative descriptions rather than quantitative results. Such results can be used as guidelines for tool development or suggest hypotheses that can be tested with a specific experiment at a later time.

Using observational techniques for systematically studying real-world processes is common. Weick documents previous uses of real-world observations as a method for retrieving meaningful contextual information about a system being studied [Weick 68]. Examples of studies that observed workers situated in their environment include domains such as airline gate assignment [Suchman & Trigg 91], air traffic control [Bentley et al. 92], subway control [Heath & Luff 92], and shipboard navigation [Hutchins 90].

Observational techniques have also been used to understand how programmers work. Approaches range from using analysis of video and transcripts to answer a very specific research hypothesis [Gray & Anderson 87, Ericsson & Simon 93] to using verbal data for exploratory understanding of planning, behavior, and problem solving.

The exploratory studies can be divided into observations in the workplace and observations in an experimental setting. Walz, Elam, and Curtis's case study of a software design team [Walz et al. 93] involved videotaping design meetings over several weeks. Berlin and Jeffries had novice programmers audiotape meetings with experts over the course of several months in order to identify the types of information experts provide that is normally not available in documentation [Berlin & Jeffries 92].

Studies involving cognitive or problem-solving issues are usually performed in a laboratory setting. The setting allows the experimenter to choose tasks and programs designed to elicit specific behavior, and to easily record observations. Letovsky observed programmers' browsing strategies [Letovsky 86] and program comprehension methods [Letovsky & Soloway 85]. Sutcliffe and Maiden explored mental behavior of analysts during problem solving [Sutcliffe & Maiden 92]. Cousin [Cousin & Collofello 92] used observations of programmers to identify information that should be provided in a software engineering environment. Lange and Rosson both documented reuse strategies in an object-oriented programming environment [Lange & Moher 89, Rosson & Carroll 93]. Other studies tested the design of a database-style programming environment for the Dylan language [Dumas & Parsons 95] and listed common problems in Macintosh programming environments [Houde & Sellman 94].

Our study is modeled on Flor's studies of organization within teams of programmers in a laboratory setting [Flor & Hutchins 91, Flor 94]. Flor's work differs from the other programmer studies because he used teams of two programmers working together as subjects. This technique, known as *constructive interaction* [Miyake 86, Wildman 95] provides a more natural way to elicit programmer talking than single-person think-aloud methods. Constructive interaction allows both observing how the programmers perform problem solving as well as exposing the possible solutions they consider by studying their dialogue.

VI.A.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 unnecessary tangents. We also found that some programmers were unwilling (as opposed to unable) to produce a program with a different structure when the directions were not specific enough. The pilot and C studies also were used for confirmation of observations in the main study.

For the final setup, we chose to instruct teams of two programmers to perform a specific enhancement to a 150 line Scheme program by using a “restructure, then enhance” process. The task was intended to be as realistic as possible while forcing the subjects to restructure by requiring a global change to a reasonably complex program. We ran the sessions in a laboratory setting to limit interruptions and facilitate video recording for later analysis. Teams worked at a single monitor. We were limited to a small task on a small program because larger programs could not be analyzed and restructured using the current restructuring tools, and because larger tasks could have taken too long to complete. Three teams finished the complete task in the allotted two hours, two finished the encapsulation but did not have time to add the enhancement, and one team did not finish the encapsulation within two hours.

Conditions. We observed programmers performing the same restructuring task under three conditions: using standard UNIX tools, using a text-based restructuring tool, and using the star diagram interface. Each team of programmers participated in one of the conditions. While the star diagram and restructuring tool teams primarily used the tools they were given, they were also free to use standard UNIX tools from a separate shell window when necessary. (Because the restructuring tools do not include comments in their presentation of the source code, programmers had to use UNIX tools at times to see the comments.) However, they 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.” By observing programmers using the text-based restructuring tool and star diagram, we hoped to identify which benefits could be attributed to restructuring, and which to the star diagram’s graphical user interface.

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 textually was distinct from methods used by other teams, and thus highlights the range of techniques programmers might apply when restructuring.

Selection of subjects. There were a total of twelve programmers working in six teams. Subjects were chosen because of their experience in programming and knowledge of Lisp-like languages. Subjects were either graduate students, many with industry experience, or programmers from local industry. All understood the concept of modularization, and all programmers had experience programming in an object-oriented language. Table VI.1 shows the variation in experience and skill of the subjects. All received a payment of $5/hour for participating. Although the money was an inducement to participate, we found all programmers to be extremely motivated, many working beyond the nominal two-hour time limit.

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. Programmers then received printed copies of the task instructions (Appendix A.A), a definition and example of encapsulation and, if applicable, a 15 minute demonstration of a restructuring tool and instructions for 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 and were debriefed.

The programmers were given an implementation of the KWIC indexing program written in a functional decomposition style (Appendix B), the same program described in Chapter II. 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

Condition, team	Subject	Experience Industry (years)	UNIX (years)	Lisp (years)	Progress in task
UNIX tools team 1	"F"	1	5	1	performed encapsulation
	"B"	0	6	5	only
text restr team 1*	"D"	7	7	6	encapsulation and
	"M"	0	6	2	enhancement
star diagram team 1	"J"	10	9	2	performed encapsulation
	"K"	2.5	6	2	only
UNIX tools team 2	"P"	7	7	10	encapsulation and
	"C"	1	6	3	enhancement
text restr team 2	"T"	0	9	4.5	encapsulation and
	"A"	4	5	3	enhancement
star diagram team 2	"I"	0	4	5	encapsulated 5 of
	"R"	16	20	4	6 functions
average/st.deviation		4.0/5.1	7.5/4.2	5.2/5.2	

Table VI.1: Years of industry experience, UNIX experience, and Lisp experience (including classroom experience) for subjects, grouped by team and tools given. * Text restructuring team 2 stopped using the restructuring tool due to frequent crashes, and instead finished the encapsulation with UNIX tools.

---

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 told 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 told to next perform the enhancement. By enforcing this two-phase process, the programmers were guaranteed to perform a separate activity that could be identified as data encapsulation.

Programmers 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.

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.

VI.A.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 ease audio analysis, we used two clip-on microphones recording to separate audio tracks of the videotape. 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. Notes written by the programmers were saved. Because the video camera was usually focused on the screen, matching notetaking with a time usually depended on context and writing noises.

VI.A.4 Analysis Method

Analysis began by transcribing each session. (See Appendix C for a sample transcript.) A transcript contains timestamped dialogue, actions on the computer, and relevant visible gestures (e.g., pointing). Because creating transcripts is time consuming, we considered avoiding full transcripts by first creating a narrative and then transcribing only the "relevant" sections. However, the narratives were not useful in focusing analysis. Our inexperience and lack of guidance from the literature probably played a role, but also restructuring decisions and the subsequent actions tended to be distributed throughout a session.

Transcripts were analyzed both "bottom-up" and "top-down" [Chi 94]. In top-down analysis, the analyst (1) formulates a specific research question ("does the star diagram exclude essential information?"), (2) identifies a question that could be mapped to the transcripts ("what information are programmers looking for when moving from star diagram to text?"), (3) analyzes the transcripts to identify and group relevant segments (called *coding*), and then (4) examines the coded segments for patterns that support or deny the hypothesis. Bottom-up analysis looks for interesting patterns and then turns them into hypotheses for top-down analysis. Some of the analysis involved explicitly coding the transcripts [Ericsson & Simon 93]; some issues involved less formal analysis of the transcripts.

VI.B Observations and Model of the Encapsulation Process

VI.B.1 Model

To understand the relationships amongst the different teams, a model of the teams' behaviors is beneficial. We analyzed the transcripts to derive a model, and discovered six kinds of actions that illuminate how encapsulation is performed, summarized in Table VI.2. The model is probably not complete. For instance, the find non-literal uses task might be generalizable to the remove obstructing structure task. Two of the more difficult tasks for this study were finding non-literal uses and choosing the functions to create, so we describe them in more detail.

Find non-literal uses task. Because this implementation of KWIC is designed using a functional decomposition, `*line-storage*` occurs literally only at the top of the program's calling hierarchy. As a result, the programmers need 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 fall into the "find non-literal uses" task. (Figure I.2 provides an example of this task.)

Choose functions task. To choose the interface functions in the `*line-storage*` encapsulation, the programmers must identify that some existing functions, such as `insline` (which inserts a line into the line storage data structure) can be used as-is in the new encapsulation. The programmers also have to choose the accessor functions for retrieving data from the `*line-storage*` list. In the study, programmers either chose to abstract away only the "lines" representation by creating a `number-of-lines` func-

Task	Subtask
Finding variables	Navigate to uses of the data structure being encapsulated. View the relevant code.
Find 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.
Choose functions to create	
Create 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.
Finish	Test done with transformations Test functionality of system

Table VI.2: Model for encapsulation. The model does not specify temporal ordering, since programmer behavior varied significantly between teams.

tion and `get-line` accessor, or chose in addition to abstract away the "words" representation by creating `get-word-on-line`, `number-of-words-on-line`, and `number-of-lines`. Either choice adequately encapsulates the representation for the subsequent change. Calls to the new functions replace several expressions directly accessing the representations occurring throughout the code. The "finding non-literal uses" task can affect the choice of functions, since not finding uses hides some computations on the representation.

VI.B.2 High-level Observations of Each Team

The following presents the observations of the six teams listed in Table VI.1. Tables VI.3 and VI.4 compile the observations for each category and shows the range of behavior we observed. To simplify presentation, we focus on the observations of the first three teams in Table VI.1; the other three teams behaved similarly, and their data is used when needed.

Behavior of UNIX tools team. F and B used `emacs` and `more` to view the code. They read linearly through the code multiple times, each time focusing on different problems. While first reading the code, they focused on understanding the code as a whole, and saw the common uses of `*line-storage*` in passing (i.e., the "find variables" task and "find 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 (grouping uses), 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 words-oriented abstraction. They then created the new functions. They then made another pass through the code to convert the expressions matching the new function bodies into calls on those functions. 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, which they reimplemented. They ran out of time before they could perform the enhancement.

	By Hand F/B	Text Restructuring M/D	Star Diagram J/K
Browsing	Searched linearly through source code, finding references by eye	Used vi's searchfunction to find each occurrence of variable in sequence	Used star diagram to direct attention to occurrences of variable used star diagram to match similar expressions
Find non-literal uses	Followed parameters to their ultimate use. Did not bother to move accesses to `*line-storage*` closer to these uses.	Used restructuring tool to inline parameters; after the crash, they performed the same change by lexical substitution.	Restructuring transformation. They noted some uses were hidden inside calls after partially restructuring.
Matching	By memory. Wrote down possible functions as found.	Identified common occurrences while searching through code. Grouped and manipulated code by regular expression matching.	Performed by star diagram representation. Programmers complained when matching did not meet their expectations.
Modifying	Created all functions at the same time, then converted all calls into the appropriate functions in one step.	Created the function declaration, then used regular expressions to find and replace occurrences of the common expressions with calls.	Star diagram performed all changes as a group. They moved completed functions into the interface to mark them as done.
Complete one modification	No such test. Created all calls in one massive set of changes.	Regexp substitutions automatically performed all changes for one modification in one step. Cleanup done by progressing linearly through file.	Star diagram performed all changes in one step.
Completing all modifications	Searched backwards by eye for unmodified references to `*line-storage*` or ls.	Searched to ensure no references to `*line-storage*` occurred outside the functions in the new module.	Worked until star diagram was empty.

Table VI.3: Comparing team behavior for subtasks of encapsulation.

	By Hand P/C	Text Restructuring T/A	Star Diagram I/R
Browsing	Searched linearly through source code, finding references by eye or using emacs	Searched linearly, looking by eye for uses of the variable. Missed occurrences.	First went through the code with Emacs, reading and understanding the code
Find non-literal uses	Followed parameters to their ultimate use. Didn't bother to move references to `*line-storage*` closer to the actual manipulations of the array	When reading through the code, they identified where the variable was passed into a call, then examined the called function.	When reading through the code, examined uses passed into calls, then transformed the code to move uses. Assumed that the increase in star diagram size indicated an inappropriate action.
Matching	By memory. When they had recognized some of the common functions, they wrote down the candidate functions. No overall view of uses.	Matched some expressions by eye. Restructuring tool automatch caused some unexpected matching because their choice of parameters was more general than they expected.	May have performed matching during passes through the code, but had difficulty identifying behavior to encapsulate
Modifying	Created all functions at the same time. Did not start adding the calls until after they had tested the created functions.	Created functions and calls with the restructuring tool, dealing with each new abstraction in sequence.	Done by the restructuring tool as a single action.
Complete one modification	No such test. Created all calls in one massive set of changes.	By memory. Memorized where replacements needed to be made. Missed some occurrences.	Done by restructuring tool.
Complete all modifications	Created each of the four functions they thought necessary, then stopped.	Created each of the four functions they thought necessary, then stopped. No explicit check that they had encapsulated all references.	Did not complete task.

Table VI.4: Comparing team behavior for subtasks of encapsulation.

Behavior of text-based restructuring tool team. 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, first trying to move nested functions apart to simplify browsing and understanding the code, then creating a function around the get-line abstraction. 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 style. Observations confirm this claim.) They used `vi` to search for references to `*line-storage*` ("find variables" task). 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 ("create function" task). In one case they used regular expression matches only to find the candidate code, making the changes by editing the file directly. After creating the functions, they found that one of their choices of abstraction, the `get-line` function, was not used outside the module, and therefore was unnecessary. The final interface included the functions for accessing words.

Behavior of star diagram team. 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 examined each node in the star diagram in turn, moving linearly down the nodes on the first level of the star diagram. They also examined the function definition nodes on the right hand side of the star diagram to identify context of the uses. They found the non-literal uses in the code. They did not appear to look deeply into the star diagram—all their functions were at the first level of 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 words-oriented abstraction, although at times they identified the more complex expressions as representing manipulations of words. The team also decided early-on to create a line-oriented abstraction, so the choice of functions may simply be due to stubbornness.

At first, J and K took the star diagram's presentation of similar expressions ("grouping uses" task) as complete, and thus delayed completing the "find non-literal uses" task with the needed transformations. When these were discovered in parts of the star diagram that they had passed over earlier, they exposed the non-literal uses with inline-parameter. Because the star diagram did not have a direct way to turn the newly exposed expressions into calls to an existing function, they backtracked by inlining the recently created function, putting all similar expressions in the star diagram together where they could be reextracted. (We later fixed this flaw by adding the "make call" transformation to the star diagram.)

VI.C Analysis of the Programmers' Modification Processes

Our videotapes show that performing the global changes during restructuring required the programmers to make changes systematically, maintaining a constant awareness of the modification's state. Each tool's support or lack of support for maintaining state constrained how the programmers could perform the modification and influenced the types of errors that could occur. In particular, we noted that the design of the tools that they employed strongly influenced how they (1) applied a consistent modification across the entire program, (2) evaluated past actions to decide how to progress, and (3) ordered tasks.

VI.C.1 Maintaining State for a Single Global Modification

UNIX tools team. (F/B) The UNIX tools team's use of `more` for searching dictated that they view the source in a largely linear fashion, scrolling down through the code. In fact, this group did not use `more`'s regular expression capability, only using its scrolling function. This approach guaranteed that they visited every use of `*line-storage*` by starting `more` on the program, inspecting each line on the current screen for uses of the variable, advancing the screen, and repeating the inspection until the end of the file is reached. This team's use of `emacs` to make changes was similar, despite `emacs`'s additional features. They applied this file-based linear strategy for all activities, including grouping uses, choosing functions, and creating calls.

The linear strategy allows the current location in the file being examined to separate variable occurrences into the set of variable uses examined (above the current point) and the set of variable uses not examined (below the current point). The programmers constantly used such an affordance of editors or browsers to ensure they examined or modified all uses of a variable. However, using the linear strategy, uses cannot be viewed together or in another logical order, such as according to the similarity of the expressions. To overcome this shortcoming, this team primarily used memorization to gain an understanding of the uses of the data structure in the code, and after making several passes over the code, they also wrote down the functions they chose to create. However, they made mistakes, inverting the order of arguments when creating calls to a function because they forgot the function's parameter order.

Text-based restructuring tools team. (D/M) The restructuring tool team (recall that they abandoned the restructuring tool after a time), extended the linear strategy by using `vi`'s regular expression matching for the finding and grouping tasks, and global substitution for the "create calls" subtask of the "create function" task.

The regular expressions they used for substitution are not trivial—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 transformational part of finding non-literal uses was also handled with global substitution, by exploiting their observation that any parameter containing the line storage was named `ls`. Additionally, when planning the substitution of `ls` with `*line-storage*`, they discovered 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. For changing `(length (line-ref foo))` to the `words-on-line` function, they searched for `''length (line-ref''`. In this case they used regular expression matches only to find the candidate code, not change it. Exploiting the naming conventions of the program to make correct changes is essentially exploiting the particular structure of the program, which goes beyond just exploiting the file structure in which the program text is embedded.

Although this group avoided F/B's problem of visiting uses out of order for the "create function" task, the uses are still physically separated in the file. Perhaps as a consequence, they employed memorization of the common occurrences of expressions during the finding and grouping tasks, and then recalled the functions to create.

Because they did not change all the uses in linear order, this group could not use reaching the end of the file as an indication of finishing the encapsulation. But they still exploited the linear structure of the file to perform the "finish" task, by searching for an unencapsulated use starting at the bottom of the new module (which is located at the top of the file):[1]

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]. And there aren't!
M: Okay.
D: That's good. So we've narrowed down the usage of that thing.

Star diagram team. (J/K) Observations of the star diagram team revealed less explicit bookkeeping by the programmers, presumably because the tool provides for explicit finding of uses, grouping of uses, and creation of functions. The "finish" task was determined by noting that the star diagram for `*line-storage*` was empty.

However, we expected to see bookkeeping in the "finding non-literal uses" and "choosing functions" tasks, since bookkeeping for these are not directly supported. Indeed, when browsing to find appropriate abstractions, they scanned linearly down the first row of children nodes in the star diagram, navigating to the text to look at details. In this pass down the diagram, they could not figure out what `csline<=` did. When they actually

[1]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.

went to restructure, they repeated the linear scanning process, but skipped over `csline<=` without mention. Presumably the linear ordering was used to facilitate organizing their plans—tracking what they had looked at and what not—and they appeared to have no trouble recalling their problems with `csline<=` and skipping over it in their plan.

In the "create functions" task we observed that although the programmers were aware that creating a function would need to be undone, they chose to create it 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.

Although it is not entirely clear from the dialogue *why* they are doing this transformation, it in part has to do with the ordering and state of the visitation, since there is a notion of consistent movement through the star diagram: "while we're here." To skip over this item without transformation might be a violation of their state-maintaining strategy. The desire to use the state of the interface list (listing and naming of operations) could also be playing a role.

Finally, the team showed consternation over the tool's failure to restructure the non-literal uses along with the literal ones. Their comments suggest that the partial book-keeping of uses provided by the tool is inconsistent with their logical view of the tool's behavior.

Discussion. All groups, in different ways, employed the linear strategy in order to keep track of what part of a task is done, and what part is yet undone, and identify when the task is complete. Apparently, the groups unconsciously prefer to visit items out of logical order (e.g., by similarity) to ensure that all items are visited. When support for matching by similarity is used, either by regular expressions or overlaying of nodes in the star diagram, other notions of linearity can be employed to organize higher level tasks. Visitation during matching and substitution in `vi` are ordered in the file, and the star diagram teams searched down the star diagram visiting each *kind* of use in turn.

VI.C.2 Evaluating Progress

Unlike bookkeeping in a specified task, evaluating progress towards the ultimate goal is not necessarily well-defined. Norman's model for planning and performing actions describes this as "perceiving system state", and "evaluating current state" with respect to the goal [Norman 86]. An evaluation may result in noting a failure in prior planning, and backtracking may be undertaken to attempt another solution. The structure provided by the tools influences how and when programmers evaluate progress.

The programmers using UNIX tools did little evaluation while restructuring, perhaps because they had already planned the action out before starting. However, there is also a case where evaluation after the "create functions" task exposes an oversight, which they choose to let go:

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.

The star diagram teams, on the other hand, appeared to regularly evaluate progress using the tool, observing the resulting code after every transformation and choosing the next transformation as appropriate. 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. This also suggests that star diagram users need access to the source code for evaluating progress.

When programmers had no obvious cues for measuring progress, they chose less desirable metrics. When I and R, the other star diagram team, performed the inline-parameter transformation to localize uses of `*line-storage*`, the star diagram's size increased (because the transformation exposed the 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 seem to realize that the code appears as expected:

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 {}

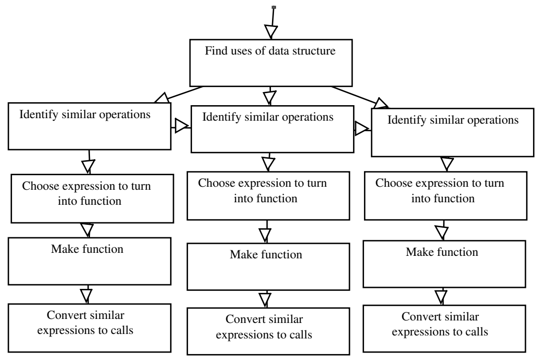
R: Yeah, definitely.

These examples show that the cues programmers use to identify progress may be different than the tool designer's assumptions. Training of tool users to avoid using inappropriate cues might help.

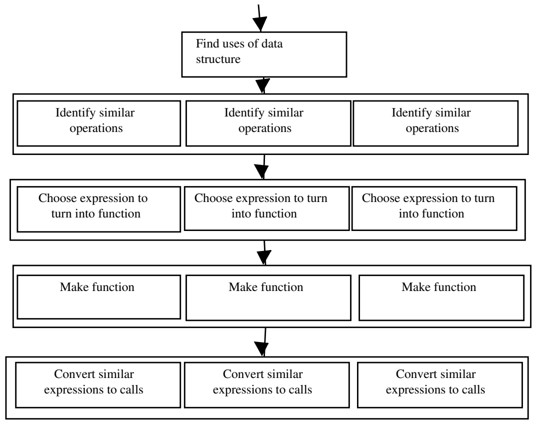
VI.C.3 Overall Sequencing of Planning Activities.

The overall sequence of activities in the encapsulation task varied significantly among the teams. These differences seem to be due in part to the different strategies used for bookkeeping.

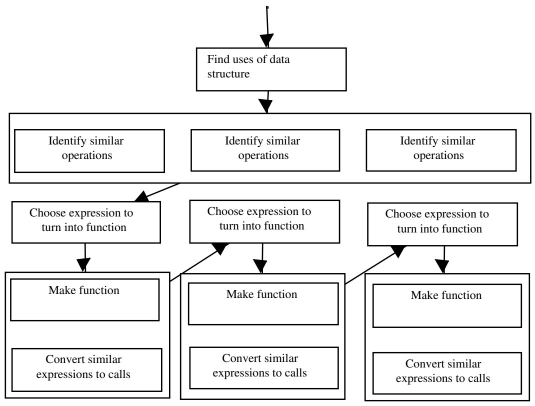
To illustrate the differences in behavior, we graphically represent the ordering of actions. Figure VI.1a shows our model of a minimal correct ordering of tasks in order to avoid backtracking in the encapsulation. For instance, we assume that the task of choosing all the functions for the interface should precede the creation.



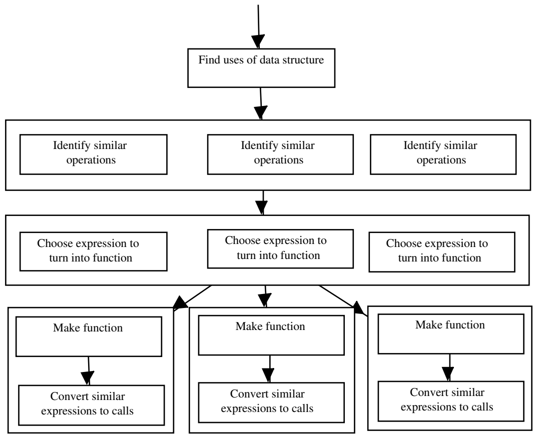
a) Dependencies in the process of encapsulation. The only dependencies assumed are that the grouping of expressions into functions is probably serialized to some extent; once likely expressions to turn to functions are found, the process of converting the expressions into calls to the new function can occur in any order.



b) Behavior of F/B mapped onto our dependency model. F/B created all functions in one set, then made a single pass through the code to convert the expressions into calls.



c) Behavior of K/B and M/D using the star diagram and text-based restructuring tool. Both teams chose functions in an ad-hoc function as they found them, and as a result sometimes made inappropriate decisions and had to back off of changes.



d) Preferred behavior of programmers. In this case, programmers carefully reason about the functions to create before manipulations.

Figure VI.1: Comparing processes used by teams. Each figure represents ordering or dependencies between actions in the restructuring process, and shows that programmers often perform similar tasks (rows) at different times when creating each abstraction or function (column).

The UNIX tools programmers F and B performed the changes in sets of linear passes through the source code, searching first to understand how `*line-storage*` was used, then passing through again to identify the functions to create, next creating the functions, then finally passing through the file to add calls to the functions (See Figure VI.1b). This process apparently reduced the number of passes over the code, minimizing one downside to a pure linear strategy. It also ensured choosing all the functions and planning the interface before creating functions. However, it separates the creation of a function definition from the creation of its corresponding function calls. Perhaps as a consequence of this separation, the programmers inverted the argument order of some calls, introducing errors.

The text-based restructuring team M/D (but working with UNIX tools) and the star diagram team J/K created functions as they noted abstractions through grouping uses, rather than deciding on the functions during initial planning. Both teams created one function, then went through and replaced expressions matching the function body with calls, then repeated the task for additional functions (See Figure VI.1c). The restructuring tool combines the creation of a function definition and creation of the corresponding function calls, ensuring consistency. However, by separating design choices temporally (because of the exploratory approach), programmers occasionally found their design choices to be inappropriate, forcing them to backtrack and undo transformations.

Originally we thought we would prefer a process like that shown in Figure VI.1d, which both chooses all functions beforehand and groups a function definition's creation with creation of its associated calls. However, the design of the star diagram does not encourage a "planning before restructuring" model. The star diagram presents the program as it currently is, and provides no support for what it might be able to look like in the future without actually changing the program. Exploring plans in detail, then, must involve performing transformations to change the source code's form. Planning in this style is possible from the star diagram and text-based restructuring tool because the underlying restructuring tool contains a multi-level undo mechanism (added after complaints in the pilot study) to allow retreating from a failed plan. All teams used this feature. Because of the inherent unvisualizability of software [Brooks 87], we postulate that interleaved planning and restructuring may be more practical than complete planning.

VI.D Validity of the Study

We can expect that the studies presented here reasonably capture the behavior of programmers in the real world. Specifically, our choice of subjects, experiment setting, and observations of programmer behavior suggest the observations are representative.

The subjects selected for the study represent the target group for programmers performing restructuring, and thus likely users of a restructuring tool. All programmers had experience in a Scheme or Lisp language, ensuring that language difficulties were not a significant source of difficulty. The programmers using the standard UNIX tools had experience both in a UNIX software engineering environment and thus understood the tools they had available. All programmers understood the concept of encapsulation and modularization which form the basis for the task. Many of the subjects had industry programming experience, and all had significant programming experience.

The setup minimized the artificiality of the study. The studies differed from a work environment in the fixed time limit, lack of access to printouts, size of the program, and group work. Although programmers in industry would not be asked to restructure a program in two hours, the task was certainly doable in the given time. Programmers in industry would also have access to additional methods of bookkeeping, such as marking printouts of the code. Although we had to explicitly ban programmers from using printouts in order to help us identify when they examined specific code, the size of the program made voluminous notetaking on paper copies unnecessary, and thus would not affect their behavior significantly. Programmers were able to mark what few notes they needed on separate sheets. The size of the program was certainly smaller than typical programs; however, the code was complex enough to limit the programmers' abilities to understand the entire program within the allotted time. Finally, although working closely with another programmer may not be a usual event, programmers do work closely together occasionally, and the experimental setup mimics such interactions.

Finally, from our observations on C programmers and from personal experience, the behavior seen with both programmers restructuring by hand and programmers using restructuring tools matches how programmers work when restructuring larger programs. However, we would expect that programmers operating on larger programs would use `grep` significantly more than they did in the current study, and as a result would gain a global, condensed view of the code. Our pilot studies show that programmers examining a twelve file C program principally use `grep` to search through the program, and do not exhibit the linear searching approaches we saw with the Scheme teams. (However, while they did not search linearly through the source code, we could expect them to proceed linearly through the `grep` output to guarantee visiting each of the matches.) Primarily, we believe that `grep` was used more in the C studies simply because of number of files and amount of code to be examined, and the programmers' realization that they could not fully understand the program in the time allotted. `grep` provides some of the features seen in the restructuring tool, such as some support for moving linearly through a set of variable references. However, it lacks methods for identifying uses of a global data structure that is passed through a function, requiring the programmers to either search for multiple variables or follow the variables by hand. We saw both styles of behavior in our programmer teams.

This use of `grep` also matches the behavior of the programmer modifying the `mbuf` structure from the UNIX kernel (Section I.A.1), where the programmer used `grep` to find variables and field names. For programs larger than tens of thousands of lines of code, our personal experience gives us few guidelines about how such programs would be restructured, but we would expect methods similar to those described in the `mbuf` example, or perhaps expect `grep` to produce too much data to be useful. Thus, we expect that while programmers would use `grep` for larger programs, `grep` would not materially change the behavior of the by-hand teams since they may still be tempted to perform changes in a single sweep through the code.

If the programmers had used the star diagram interface on larger programs, we expect that their behavior would have been similar to the behavior shown in this study. The most significant change would probably have involved problems with browsing through larger star diagrams. Chapter V addresses some of the difficulties programmers restructuring a large program through a star diagram may have, and suggests how adjustments to the star diagram visualization could support large programs.

For these reasons, the observations presented here appear to be representative of how programmers restructuring large programs would behave.

VI.E Summary

Little is known about how programmers restructure programs. We applied 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.

All teams were capable of restructuring the programs to encapsulate a data structure. Also, all teams carried out six basic tasks to accomplish the restructuring, suggesting that there may be a common underlying basis for encapsulation activities.

The structure provided by each of the tool sets strongly influenced how the restructuring task was performed and what mistakes they made. In particular, three different levels of activities—maintaining state for a particular change, evaluating progress of the overall change, and choosing an ordering for the overall change—were carried out very differently. These differences appear to be related to the way these tools maintain state for the programmer. Most notably, much state information that is not explicitly maintained by a tool is managed by the programmer by using a linear strategy of visiting nodes in the file. Such a strategy ensures that all relevant instances are visited, avoiding errors of omission. However, this comes at the expense of visiting those instances in an order non-optimal for certain tasks, and can result in errors of recall as programmers rely on memory to identify similar expressions.

> 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."

In working with UNIX tools, which largely lack explicit bookkeeping features, teams made errors of recall that resulted in errors of functionality and failures of design. In working with the star diagram, which employs several bookkeeping features, teams cannot make errors in functionality, but still experience failures in design. Also, the star diagram's bookkeeping features, such as meaning-preserving transformation, multi-level undo, and the interface function list seem to encourage an exploratory style that results in backtracking (i.e., failures) in design. We hypothesize that such a style is appropriate for larger software systems, where *a priori* visualizing and planning a complete encapsulation may be uneconomical.

Chapter VII

Related Work

Previous work that inspired the star diagram or complements the star diagram concept falls into two major categories: determining the "relatedness" of expressions in the code, and visualizing and manipulating software structure.

VII.A Identifying Relatedness of Nearby Expressions

Other methods exist for identifying the "relatedness" of two expressions. One of these approaches is the concept of coupling and cohesion, a descriptive term that has been used to create metrics that measure the quality of the software structure. The second approach to identifying relatedness uses dataflow analysis to identify the statements that, because of computation patterns, are dependent. The technique of slicing, designed to provide a logical view of a source of bugs, is one such use of dataflow analysis.

VII.A.1 Coupling and Cohesion

Coupling and cohesion are two subjective criteria that can be applied to a system when evaluating its structure [Stevens et al. 74]. The cohesion of a program component is a measure of how related its subcomponents are. The coupling between two components is a measure of how related they are. (It is not always clear whether a relationship is coupling or cohesion, because it is sometimes subjective whether two components are subcomponents of a module or separate components.) In general, high cohesion of a component is desirable, because when one of its subcomponents must change, the other subcomponents are likely to require changes for the same reason. Consequently, automatic computation of cohesion and coupling measures has been investigated as a way of assessing modularity, perhaps for driving manual restructuring. Most approaches are based on clustering or slicing.

Schwanke assumes that procedures referencing the same name must share design information on the named item, and are thus "design coupled" [Schwanke 91]. He uses this concept as a clustering metric to identify procedures that should be placed in the same module. This approach places no special emphasis on types or names of data structures.

In contrast, Embley and Woodfield's clustering method focuses on the types and operations exported from a module and the types used within a module [Embley & Woodfield 88]. The method identifies modules that act on more than one abstract data type or act on only subparts of a type, thus suggesting poor cohesion or inappropriate coupling.

The approach of Ott and Thuss [Ott & Thuss 89] does not search for relationships among components that are dispersed throughout the program, but focuses on the statement relationships in a procedure via slicing [Weiser 84]. In this approach, the computations that produce a particular value are related through the value. Conversely, computations that are not working together to produce a value are unrelated. In this sense, slicing seems to capture communicational and functional cohesion. For example, when the slices of two output (reference) parameters in a procedure do not intersect, then the computations that produced them are unrelated with respect to slicing, and so are less likely to be communicationally or functionally cohesive tasks.

The star diagram displays relationships that suggest several of the commonly recognized kinds of coupling and cohesion. First, the fact that all the nodes in the diagram pertain to a single data structure indicates design coupling or functional cohesion, depending on the context. The repetition captured by node stacks also suggests design coupling. Second, the data flow and control dependence relationships of the star diagram, which are largely limited to within a function definition, suggest communicational or functional cohesion. Intuitively, those connections closer to the root of the star diagram are more strongly related to the maintenance of the data structure, and hence are more likely to be functionally cohesive. Consequently, an edge identified by the tool user as representing the weaker communicational cohesion might imply that the more strongly cohesive expression represented by the path to the left should be extracted as a function, so that the result of the call represents the communicational relationship. Third, the scoping information provided by the function definition nodes and the link to the text view provide insight into temporal and sequential cohesion. Although the relationships between the computations in a star diagram are implied to be strong, weaker but still useful forms of cohesion and coupling can be suggested by our enhanced text view that layers data flow onto the text view (Section IV.A.2).

VII.A.2 Relationship of Star Diagram to Slicing

The star diagram assumes nested computations represent a cohesive computation, and that nested expressions connect these cohesive components. Sometimes, the cohesiveness of two statements cannot be identified using only syntactic information. For example, when a value is saved into a temporary variable, the star diagram uses dataflow information to identify where the value is later manipulated. Other program visualizations, such as *program slicing* [Weiser 81], also use dataflow analysis to identify related statements in a program.

Program slicing was originally intended to help programmers find bugs in programs. Given a variable use with an incorrect value, a programmer creates a "slice" of that program—a subset of the statements in the program that could affect the value of that variable. Calculating the slice requires finding all statements reaching the variable use by dataflow or conditionally affecting its execution (See Figure VII.1). Weiser's original definition of slicing identified the set of statements that could affect the value of a variable. These have been described as "forward slices" [Horwitz & Reps 92], while slices that indicate all statements that could be affected by the value of a variable have been called "backward slices". More recently, slicing has been proposed as a tool for helping programmers understand large programs.

Like slicing, the star diagram uses dataflow to identify statements that are part of the same calculation. Because a backwards slice represents a minimum subset of a program that contributes to the calculation of a variable's value at a given point within a program, a slice is immediately recognizable as a cohesive set of statements. The slice can be calculated by identifying all statements reachable through control or data dependences from the slice point. The star diagram similarly assumes that statements connected via data or control dependences are likely to be part of the same concept or calculation.

The star diagram differs from slicing because of the star diagram's focus on relating, comparing, and manipulating similar but distant code instead of consecutive statements in the source code. By grouping statements scattered throughout the code, the star diagram simplifies planning and performing the encapsulation. Because the star diagram algorithm (unlike slicing) does not follow dataflow outside a function when creating the star diagram, the star diagram avoids presenting cohesive statements that may not be able to be moved together due to dataflow constraints, and avoids performing transformations that may upset the programmer's understanding of the code too significantly. By only matching and drawing the similar aspects of code, the star diagram sacrifices understanding individual statements in exchange for comparing distant, similar code. By focusing only on the dataflow generated by return values, we implicitly decide that the most important measure of similar statements is a similar chain of operations in the code. This simple method of comparison also ensures that the programmer looking at a star diagram can quickly perceive from the star diagram what a node represents. Looking primarily at dataflow generated by return values of functions also means we can generate the star diagram with only the abstract syntax tree. All these decisions ensure that the star diagram supports the task of encapsulation, can be created quickly for large programs, and allows nodes in the star diagram to represent expressions the programmer will wish to manipulate.

The concept of "forwards" and "backwards" slices poses the question of whether a star diagram that followed dataflow backwards could help encapsulation. With our current forward dataflow, we ignore the expression on the right hand side of an assignment to the

For the cssetup function in KWIC:

```
(define cssetup (lambda ()
  (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)))))))
```

The below code shows a backwards slice on the assignment to numwords. We have left out the irrelevant computations.

```
(define cssetup (lambda ()
  (let ((numcslines (allwords *line-storage*))
        ...
        (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)))
        ...))))
```

Figure VII.1: Example of a program slice.

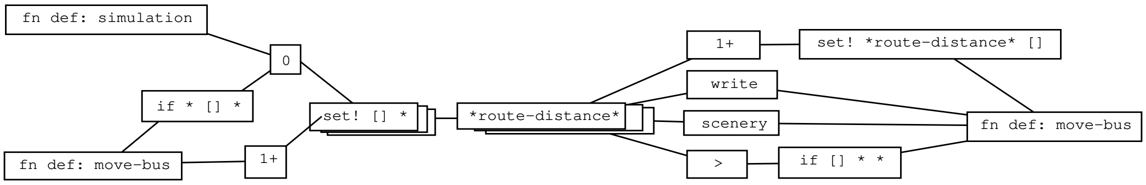


Figure VII.2: Possible appearance of a star diagram for `*route-distance*` where assignments and reads are separated. This represents the `bussim` program introduced in Chapter 1. The root of the star diagram appears in the middle of the diagram.

variable being encapsulated. In the expression

```
(set!  *line-storage* (list numlist (+ 5 2)))
```

the `list` expression does not appear in the star diagram. However, we could try to represent backwards dataflow in the star diagram to show from where assignments to `*line-storage*` get their values. One possibility is to divide the star diagram between variable uses and variable assignments, and draw separate star diagrams for each. In this way, programmers could see similarities between both types of uses. Figure VII.2 suggests the appearance of such a star diagram.

VII.B Visualizing and Manipulating Structure

The second major category of related work represents approaches for restructuring. First, a number of other systems visualize software structure in order to help the programmer reason about software. Second, a number of other approaches exist for manipulating programs.

VII.B.1 Visualizing Structure

There are many systems that support working graphically with programs; most provide little support for manipulating programs. We discuss a few that are most relevant to our work.

The Rigi reverse engineering environment provides semi-automatic construction of system views that assist understanding [Muller et al. 92]. Views are initially constructed by the environment and then modified and augmented by the environment user. These additions by the user allow describing the system in abstractions not visible in the implementation. The combination of automatic construction of views, combined with user customization, supports quick construction of views with complete user control. Rigi uses spatial and visual display of information to facilitate rapid understanding. Spatial display reveals contextual relationships, and visual differentiation and connection reveals similarities of components and their non-spatial relationships.

The star diagram would benefit from a similar visual model that supports customizing the star diagram display and collapsing of unimportant information. The star diagram currently provides little such support. Rigi's ability to augment views can be used in our tool for eliding currently unnecessary detail to provide more space to the part of the system under scrutiny. Rigi's manipulations, however, only augment the external descriptions of the system, and do not allow manipulation of the implementation directly. To support manipulation of the underlying program, additional representations must be added to the tool to provide the program text in a manipulable form, and detailed connections between the design and implementation representation are required. Rigi currently only analyzes the text to the extent required to derive the high-level design relationships.

The Garden project [Reiss 87] notes that programmers benefit from using many notations during design, each view suited to a design technique. The STATEMATE environment, for instance, supports this need by providing physical layout, dataflow, and state machine notations to describe and prototype reactive systems [Harel et al. 90]. Garden permits the programmer to define custom textual and graphical notations and describe designs with these notations. Garden then keeps the multiple views of a design consistent with each other as manipulations are performed, and supports limited rule-based execution of the designs to permit testing certain properties.

Garden does not require meaning-preserving transformations or links to the implementation because it focuses on designing new systems, rather than maintaining existing ones. Since no implementation exists in the initial design phase, maintaining correctness of the design only requires comparing the programmer's notion of correctness against the execution of the design. If Garden were used for manipulating the design of existing systems, additional transformations that preserved properties of both the design and implementation might be defined. Garden's observation that multiple design notations are necessary during design suggests a need for multiple notations to aid restructuring in our tool. For example, the notation we use can highlight data-encapsulated structure. Another quality of the design, such as layering, might benefit from a customized notation displaying layering properties.

The PegaSys system is a program analysis system based on visual formalisms [Moriconi & Hare 85]. Although the formal verification of program properties is valuable, the work involved in dealing with textual formalisms and relating them to the program implementation is daunting. PegaSys adds a layer of visual formalisms on top of textual ones to speed constructing a description and facilitate understanding. Visually, these formalisms are not unlike the ones provided in Rigi, but are perhaps are at a higher conceptual level. PegaSys also provides a straightforward graphical interface for the tool user to manually use of user input to connect design elements to implementation is a technique that we use in our system for transformations that create new abstractions. PegaSys could probably ease the connection task with the semi-automated connection technique supported by Rigi.

VII.B.2 Other Approaches to Restructuring

Although the star diagram solution matches our expectations for a restructuring tool (Section II.A), other approaches to resrtructuring could ignore some of our assumptions. For example, other approaches could restructure programs with simpler, syntactic transformations, could manipulate programs via specifications, not implementations, or restructure without programmer control.

First, other examples exist of tools that allow the user to plan how the restructuring should proceed. Srinivasan's restructuring tool [Srinivasan 93] remodularizes a program using a clustering metric focusing on the names of data structures. Restructuring occurs by first removing existing structure, then creating new modules to encapsulate sets of data structures specified by the tool user. For each chosen set of data structures to put in a single module, all procedures directly accessing the data structures are placed into a module. Any procedures that cannot be moved (such the main procedure or a procedure that has already placed in another module) are replaced with calls to a reader or writer function. Like the star diagram, Srinivasan's tool creates modules based on the named references to the selected data structures. Srinivasan's approach is simpler than Griswold's, since it is basically syntactic, requiring no semantic analysis. However, Srinivasan's approach assumes that the proper functions for the module's interface already exist in the program being restructured, whereas the star diagram and underlying restructuring tool allow the tool user to create an appropriate set of functions from the code using the data structure.

Second, the programmer could be presented with abstract views of the source code that provide a global view of the program. The programmer could then directly perceive how the data structure is used, and plan the restructuring from this level. Such an approach could be at the level of design diagrams, or at the level of specifications. Transformational methods have proposed that modification is best performed at a specification level. Transformational systems let the user describe the program as a specification that is semi-automatically made concrete by a series of specification-to-implementation transformations. Modifications can be performed by reasoning about and changing the specification, then allowing the transformation system to automatically propagate changes down into the implementation automatically [Bull 90, Balzer et al. 76]. In such a system, the programmer could reason about a restructuring as a change to the specification (for example, stating that the instantiation of two variables should occur in the same scopes), and then let the system rederive the implementation.

Finally, Griswold's work instantiates a very specific policy: because the restructuring tools are designed for simplifying an immediately forthcoming maintenance task, the programmer must be in control of restructuring. Following this course would suggest minimal support for automation, but more support for identifying how the design should be changed. Alternatively, various systems exist that could make guesses at interfaces. Srinivasan's restructuring tool followed the simplest automated method by only encapsulating variables with accessor functions. Various AI approaches could keep schemas describing possible interfaces, and a tool could automatically match an appropriate interface to the code.

Rather than help the programmer visualize and understand the entire structure, automated tools could perform the understanding and restructuring steps. Programmer feedback in such a tool could be minimal (similar to the goto removers of the past [Bush 85, de Balbine 75]) or significant. By recognizing concepts and identifying appropriate changes, such tools could restructure the program automatically, or present suggestions to a programmer [Biggerstaff 89, Quilici 93, Soni 91]. Alternatively, like an old version of the restructuring tool, the programmer could identify a common concept, then let the restructuring tool find and restructure similar code.

VII.B.3 Approaches for Supporting Encapsulation

Several approaches could help the programmer restructure. First, we could choose to weaken the design restriction that the programmer be in control of planning the encapsulation. Automatic techniques, like those used for restructuring, could be applied to encapsulation. As mentioned before, automating the task removes the programmer from decision making. One advantage of encapsulation, however, is that the task might be constrained enough to allow good automated decision making. A number of metrics exist for identifying cohesive statements in a program; such methods could be used to identify candidate interface functions for an automatic encapsulation tool [Ott & Thuss 89, Schwanke 91]. By recognizing concepts and identifying appropriate changes, such tools could restructure the program automatically [Biggerstaff 89, Quilici 93]. Alternatively, a restructuring tool could let the user specify a pattern describing expressions to extract, then allow the tool to automatically match and restructure similar expressions. Encapsulation tools could also rely on programmer input for the more difficult decisions, and automate the simpler tasks. For example, Srinivasan's modularization tool allows a programmer to encapsulate a variable or data structure by choosing the existing functions that should be in the module encapsulating the variable. Once the programmer has chosen the functions to move into the module, the tool automatically converts any remaining direct accesses to the variable into functions that set or retrieve the value of the variable [Srinivasan 93]. A restructuring tool can also present a list of possible transformations and let the user control the tool's overall behavior [Soni 91].

VII.C Summary

The star diagram demonstrates that programs can be restructured by manipulating a meaningful graphical representation of the program. Like many complex engineering systems, the design of this implementation required a number of tradeoffs in terms of how we identify related program components, what restructuring task we perform, how we visualize the program, and how we restructure the code. The related research shows some of the alternative choices that could have been made. Rather than inferring dataflow relations from the AST, we could use any of the other cohesion metrics, or use a sliced representation of the source code to discover cohesive expressions representing single computations on the data structure being encapsulated. Rather than use a restructuring tool to actually manipulate the code, we could use less expensive methods such as Srinivasan's restructuring tool, or use the star diagram to restructure specifications. Alternatively, the star diagram itself could serve as a rough metric. An automated restructuring tool could encapsulate a variable by following basic rules about restructuring operations to perform based on star diagram appearance. These works thus contribute to the range of possible implementations to which the star diagram concept can be applied.

Chapter VIII

Conclusion

Encapsulating a data structure in a program is a difficult and error-prone process requiring the programmer to have a global understanding of how the data structure is manipulated, and to understand in detail the contexts in which the data structure occurs.

This thesis studies the question "how can a restructuring tool guide the programmer through the planning and execution of encapsulating global variables?" We introduced a manipulable program visualization called the *star diagram*. The star diagram both highlights information pertinent to encapsulation and supports the application of meaning-preserving restructuring transformations on the program through a direct-manipulation user interface. The visualization graphically and compactly presents all statements in the program that use the given global data structure directly or indirectly, helping the programmer to choose the functions that completely encapsulate it. The star diagram's tree structure allows the programmer to quickly identify similar expressions, and note how they differ. Additionally, the visualization elides code unrelated to the data structure and to the task, and collapses similar expressions to allow the programmer to identify frequently occurring code fragments and manipulate them together. The visualization is mapped directly to the program text, so manipulation of the visualization also automatically restructures the program.

We have shown that the star diagram concept assists programmers in restructuring by first implementing a star diagram view on top of an existing Scheme restructuring tool, and showing that the star diagram can be implemented (Chapters II and III). By implementing star diagram generators for C and MUMPS, we showed that the concept can be applied to programs in other languages We investigated the scalability of the star diagram by creating and examining star diagrams for variables in large C programs (Chapter V). Finally, to see if the star diagram can actually be used by programmers, we studied how programmers use the star diagram to manipulate programs, and found that the star diagram significantly changes the process that the programmers follow (Chapter VI).

VIII.A Open Issues

Although this dissertation has discussed how the star diagram can be used in real settings, a number of difficulties with the star diagram remain unresolved.

Use of Star Diagram without Restructuring Tools: The current star diagram relies primarily on a restructuring tool to perform the actual modifications of the code. Without a restructuring tool, is the star diagram still useful? We believe the star diagram shows promise even in a non-restructuring environment. We have seen in examples of star diagram use (Sections II.D and V.A.2) that the star diagram may help with program understanding. We are also encouraged by using the star diagram to identify bugs in programs. When we examined the star diagrams for `adventure` (Section V.A.2), we were able to identify assignment had been used instead of equality test in an expression. In the `emacs` code, we noted cases where a static computation was performed inside a loop, instead of outside it.

Even in a non-restructuring environment, the star diagram might also provide minimal support for restructuring and helping the programmer identify abstractions. The tool user could manipulate the star diagram to plan a design, then receive a list of the transformations performed and suggested details on what lines of code to change [Chen ss].

Extracting Multi-statement Functions: The star diagram currently does not support either identifying or creating functions made up of multiple syntactically independent expressions or statements. To help the programmer identify blocks of statements that occur throughout the code requires the restructuring tool to carefully identify relevant fragments through the code and present perhaps hundreds of blocks of code in a clear manner (Section IV.A).

Some method for identifying conceptual blocks of statements is necessary. The approaches we proposed, such as analyzing the cohesiveness of blocks of code to find sets of statements tied to a single purpose, using informal hints in the code to identify chunks of code, or using artificial intelligence techniques for finding blocks of code representing common operations, could attack this problem. The star diagram can then represent similarity of concepts in the program rather than just similarity of operations and expressions.

Visualizing, Comprehending, and Manipulating Large Star Diagrams: Although Chapter V shows that usable star diagrams can be created for large programs, we also find that reducing star diagrams to a single screen or page may be unreasonable. except for the simplest variables. As a result, techniques for helping the programmer navigate the star diagram and identify statements of interest is essential for the star diagram to be practical.

Parsing: Because the star diagram requires a parsed version of the code, the source code to be analyzed must be reasonably complete and without syntax errors. Parsing, although not expensive, still carries a finite cost, and so for million-line programs, the delay in creating the star diagram for a single variable may be excessive for helping the programmer answer specific questions or plans. We could avoid these difficulties by maintaining a parsed version of the program that is constantly updated, and thus available for quickly building star diagrams. However, programmers restructuring large programs may work more methodically, and so may not need new star diagrams to appear instantaneously.

Need for Multiple Views: Because the star diagram visualization is focused towards a specific task, it can only be useful for a class of restructuring problems. The star diagram, by providing a highly focused view, introduces the need for additional focused views, each focused towards a common task the programmer may encounter. We have seen that the structure diagram assists tasks that the star diagram does not, while the star diagram assists tasks that the structure diagram cannot (Section II.F.3).

The effort of creating each of these specialized views, focused towards every task a programmer could wish to perform, could prove too expensive relative to the benefits provided by the most useful visualization of the set. As a result, we may need to identify a small number of views on the program could support all the restructuring tasks that a programmer might encounter, easily support the creation of new manipulable views, or assume that programmers will sometimes be forced to restructure via the source code when no specialized view is available.

VIII.B Contributions of the Research

This work presents an approach to helping the programmer plan the restructuring of a program. The star diagram can help programmers perform encapsulation by presenting the details of the program relevant to encapsulation, and allowing the programmer to perform restructuring changes directly from the star diagram.

The work also provides a number of specific contributions:

Star diagram concept: Primarily, we found that the star diagram concept can help a programmer plan and perform encapsulations by viewing a program in terms of the uses of a data structure and grouping that code by similar behavior, not viewing the program by file and grouping expressions according to where they appear. Our implementation of the star diagram in the Scheme restructuring tool shows that the approach is feasible, and does in fact support the task of encapsulation. We provide a generic algorithm for generating star diagrams that allows the concept to be tailored to specific languages, encapsulation tasks, or domains.

Our implementation of the star diagram for C and MUMPS shows how the star diagram concept can be translated into other languages, and suggests approaches for using star diagrams for additional languages such as assembly language (Chapter V). By creating star diagrams for large C and MUMPS programs, we also showed that the techniques can be scaled to million-line programs with appropriate support for focusing the programmer's view of the visualization.

Understanding Restructuring and Encapsulation: Our work increases our understanding of restructuring. Encapsulation can be seen as a driving paradigm for restructuring because it provides a strongly motivated task (creating abstract data types) and provides a direct method for simplifying any change: "localize what's going to change." Thus, encapsulation can focus a programmer's view of a specific change.

We also see that the dominant problem with restructuring is that the program's form changes quickly as the programmer restructures. A visualization such as the star diagram always displays the program in a form suited to the task, and so keeps the programmer informed about the current form of the source code.

Low-cost Cohesion Analysis: Our implementation of both C and Scheme star diagrams also shows that imprecise dataflow information can be gathered from the AST (Chapter III). The star diagram shows that a simple cohesion metric such as "enclosed in" and "same operation as" can identify tightly cohesive statements. As others have shown, the parsed source code allows quick analysis for some dataflow and control flow [Cousot & Cousot 77, Ballance & Maccabe 92]. The star diagram presents a case where the imprecise dataflow enables us to generate abstractions of the code that are meaningful to the programmer for the task of encapsulation.

Focused Visualizations Support Restructuring Software contains multiple constraints and connections between statements, and is often described as "unvisualizable." [Brooks 87] However, the star diagram succeeds as a visualization primarily because it is a focused representation intended to help the programmer with one specific task. Although the star diagram may not display all the information the programmer is concerned with, it presents the most important: statements using a given variable, and the similarity between the containing expressions. More importantly, because the star diagram is implemented on top of a meaning-preserving restructuring tool, the programmer using the star diagram can ignore certain constraints in the source code, such as data flow and control flow, because the restructuring tool guarantees these other details remain constant. Finally, the star diagram does not need to present all relevant information about the encapsulation that the programmer may need because all nodes in the star diagram are directly linked to a view of the corresponding source code. The combination of a focused view of the program relative to a specific task with the ability to ignore details of program behavior allows the programmer to work primarily at the level of the visualization, and thus shows that programs can be manipulated through graphical forms.

The star diagram is not a solution for all restructuring problems. Some tasks—such as changing the scopes of functions—are better performed in a visualization such as the structure diagram (presented in Section II.F.3) which explicitly represents scopes in a manner more suited to the task.

How Programmers Plan and Perform Restructuring: The studies of programmers provide three major contributions concerning how programmers restructure (Chapter VI).

- First we found a set of tasks that captured the programmers' behavior during each study. By identifying these tasks, we can now use the framework to evaluate and design other tools for restructuring.

- Second, the affordances that a programmer's tools provide, such as encouraging the examination of each use of a variable in sequence, significantly affect the manner that programmers perform encapsulation, and thus affected the kinds of errors they made. For example, programmers modifying the program with a text editor used affordances of the text editor that encouraged passing linearly through the code to make all modifications at once. This feature of the text editors encouraged converting all uses of the `*line-storage*` variable at once. As a result of separating the creation of all functions from the creation of all calls, the programmers swapped the order of parameters in some function calls.

- Third, the studies showed that the star diagram provides support for bookkeeping and planning not found in the other tools. However, its support for manipulation and bookkeeping encouraged an exploratory, backtracking design style that avoided long-range planning. Tools that supported more limited bookkeeping support (such as using the standard UNIX editors and searching tools) led to more planning. However, without the bookkeeping support, their plans suffer because the processes encouraged by the tools also cause programmers to perform tasks in an order that required memorizing or recording details to avoid errors.

Constructive Interaction for Evaluating Software Tools and Processes Although testing specific features of a complex software engineering tool may appear logical, simple, and cheap, any tool used in a complex task will be strongly affected by the surrounding environment and the normal working habits of the programmers. By using observational methods in a pseudo-realistic environment, we observed exactly how programmers performed restructuring, both and without automated tool support, and as a result identified that programmers required support for bookkeeping and maintaining state. The methods do not necessarily prove the benefit or cost of a given feature, but presents the design tradeoffs and features most important to the tool. The observations thus allow us to better design future software engineering tools to help the programmers to maintain awareness of the state of a modification in a complex restructuring task.

We are pleased with the issues identifiable through detailed observations of programmers in realistic situations. The constructive interaction technique proved extremely useful at eliciting programmer thought processes with minimal training. The methods facilitated identifying the processes programmers followed during encapsulation. Although examining programmer behavior at such a low level may appear time consuming, we would not have seen the wide variation in how programmers perform common tasks using less expensive methods.

Appendix A

Experimental Instructions

A.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 of 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.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.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.

A.B Instructions for the Star Diagram

- Introduction
- Instructions for main text view window
- Star diagram Key
- Picture of a star diagram
- Star diagram actions and transformations
- Things to remember about ERNIE

You have access to a tool called "ERNIE." "ERNIE" helps you do data encapsulation by

- helping you decide how to encapsulate a variable
- automatically restructuring the program based on your decisions
- making sure that no bugs are introduced into the restructured code by ensuring that the code executes the same before and after restructuring.

A.B.1 Process

ERNIE works best if you plan your changes using a restructuring process. Enhancing a software system by restructuring means that you break the modification up into two parts:

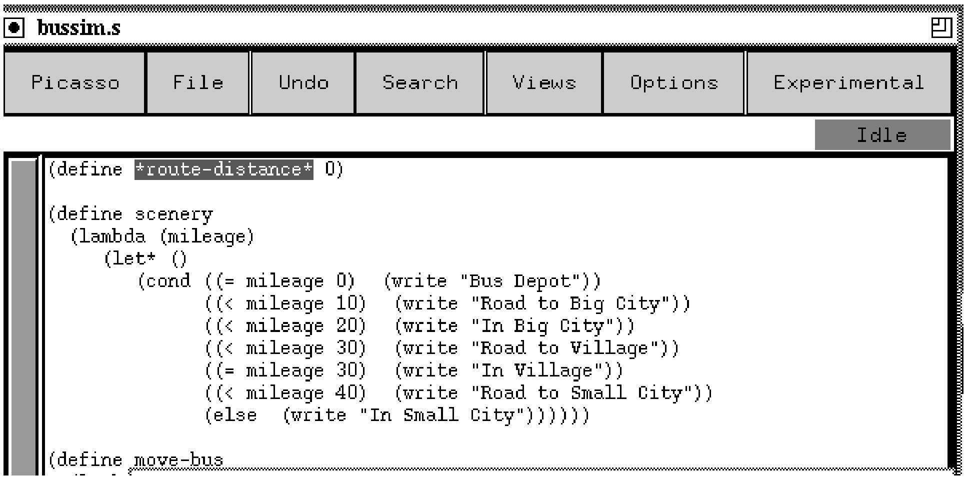
Given a program where the change will be hard to make,

1. Change the software's structure so the modification can be added easily. For example, encapsulating a data structure inside a module lets you change the data structure's behavior by changing only the functions encapsulating the data structure. By using ERNIE, you can update the program's structure without changing the code's behavior, and so you know you're not introducing bugs.
2. Next, add the enhancement. If you did the restructuring right, the modification only involves the module/set of functions that you created during restructuring, because you'll have less code to change and you'll only need to test the changed code.

ERNIE specifically helps you encapsulate a global variable by restructuring the program and helping you create the set of functions that hides the data structure from the rest of the program. ERNIE makes sure the change won't affect the running behavior of the code, and ERNIE then restructures the source code of the program, changing the source but not changing the running behavior of the code. Once ERNIE's restructured the program, you should be able to add your modification to the program easily.

Later pages describe the transformations you can perform on the program.

A.B.2 Instructions for Main Text View Window



Command summary

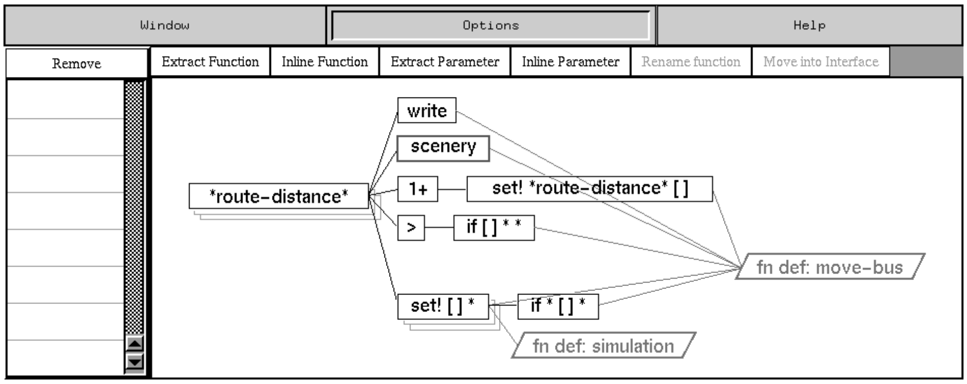
The text view controls major functions of ERNIE. The menus allow you to perform major operations. The Idle Indicator tells you when ERNIE is still performing a transformation. The text view below lets you see what code is associated with a node in the star diagram (click on a node in the star diagram to see this). Also, the source code window allows you to click on text to highlight expressions. Clicking once on an unhighlighted piece of text highlights the item underneath the cursor. Clicking on a highlighted item causes the containing item to be highlighted.

Here is a list of menu items in the text view. Each menu item is followed by a description of the command performed when the menu item is selected. Items marked Ignore are unneded during the study.

Menus in the Text Window

Picasso:	
Quit	Stop the application. Unneeded.
File:	
Info	Program name, authors. Irrelevant.
Load	Load a new Scheme program. Don't use, the file's already loaded, and loading a new file could take 10 minutes.
Edit Program	Don't use. Save the file out and edit it as usual.
Run Program	Don't use.
Print	Prints the source code.
Save	Save the current source code out to the file of your choice.
Undo:	
Undo	Undoes one previous transformation. You can perform undo multiple times in order to undo a sequence of transformations.
Search:	
Next selection	Moves to next highlighted item of a set of highlighted sections
Prev selection	Moves to previous highlighted item of a set of highlighted selections
Search forward	Lexical search for arbitrary string. Next/prev doesn't work on this.
Query	Ignore
Slice	Ignore
Variables	Highlights all occurrences of a given variable name.
Views:	
structure diagram	Ignore
star diagram	As seen in the demo. Helps you encapsulate a global variable.
Options:	Ignore this menu

A.B.3 Key for Star Diagrams



The star diagram has four main regions: the *menu bar*, which contains commands for manipulating the star diagram; the *transformation buttons*, which allow you to manipulate the program; the *interface function list* for keeping track of the functions you want to encapsulate the global variable, and the star diagram graph itself.

Star diagram graph

The *leftmost box* (the root) represents all uses of the variable in the program.

First level: Operations directly referencing the variable are represented by nodes connected to the root.

Second and further levels: Operations consuming the result of a first level operation are connected by an edge.

Boxes represent an expression in the code, and the label on the box represents the operation or function being performed. For some Scheme operations (`set!`, `if`), the label indicates the location of the variable use. `[]` indicates the contained expression (the node to the left), and * indicates a "don't care" placeholder. So, `set!  [] *` indicates the variable occurs on the left hand side of the assignment, and `if * * []` indicates the variable occurs in the `else` clause of the `if`.

A *bold box* indicate the operation is a user-defined (as opposed to built-in) function.

Parallelograms indicate the function the expressions are contained in.

Stacks of nodes: when two nodes denoting the same operation are connected to the same node on the left (starting at the root), they are stacked on top of each other, thus indicating the expressions representing the expression are similar. A stack of two nodes indicate two similar expressions in the code; a stack of three nodes represents three or more similar expressions. Nodes are similar when the operations from the root to the node are similar.

Lines connect operations where the result of the left operation is used by the right operation.

Black lines indicate that the operations are nested.

Purple lines indicate that the operations are connected by dataflow, not by nested.

Blue edges connect an expression node to the function definition that the expression occurs in. The blue edges give you an idea of the context the expressions occur in.

A.B.4 Star Diagram Actions, Menus, and Transformations

Actions:

To view code associated with a star diagram node:

- Click on a box in the star diagram with the LEFT mouse button
- The code represented by the box in the star diagram will highlight in the text view.

To adjust star diagram image:

- Move the image by pressing CONTROL and MIDDLE mouse button, then moving the mouse.
- Zoom in on a part of the image by pressing CONTROL and LEFT mouse button, then dragging across the region to zoom in on.

To restructure a program

- Click on a box in the star diagram with the LEFT mouse button to select the code to manipulate
- Click with the LEFT mouse button on one of the restructuring transformations
- Answer questions in dialog box to choose details of transformation.
- After the transformation, the source code and the star diagram will be updated.

Star diagram window menu commands

Window:	
Show all	Make star diagram image exactly fill the window.
Print...	Ignore.
Done	Closes window
Options:	ignore
Help:	
Star diagram	Shows a short description of the star diagram

Star diagram restructuring transformations

Extract function	Inline function
Extract parameter	Inline parameter
Rename function	Move into interface
Remove (move out of interface)	

Extract-function: Given selected node representing single statement (or multiple similar statements) in the code, convert the statement into a call to a new function, and make the body of the new function be the same as the statement. You'll need to name the new function and choose parameters for the new function (which also decides what part of the statement(s) are inside and outside the new function).

Example: With the code:

```
(foo 5)
(define foo (x) (+ x 10)),
```

Extracting (+ x 10) as a new function add-ten causes the code to look like:

```
(foo 5)
(define foo (x) (add-ten))
(define add-ten (num1) (+ num1 10))
```

Inline-function: If you've selected a function call node (bold box, as opposed to a system calls), inline replaces the function call with the body of the function, thus removing the call. If the function's not used anywhere else, the function is assumed to be unneeded and removed from the program.

If you've selected a function definition node (fn-def box), then inline replaces all function calls with the body of the function, then, since the function is not used anywhere, ERNIE removes the function definition from the program.

Example: Inlining foo in the below code

```
(foo 5)
(define foo (x) (+ x 10)),
```

causes the program to be changed to

```
(+ 5 10)
```

Extract-parameter: Given a selected node representing a statement or expression in the code, make the expression into a parameter to the function, moving the computation outside the current function.

Example:

Doing extract-parameter on the number 10

```
(foo 5)
(define foo (x) (+ x 10))
```

causes the code to change to

```
(foo 5 10)
(define foo (x new-var) (+ x new-var))
```

Inline-parameter: Inline-parameter is the opposite of extract-parameter. Given an expression passed into a function call, replace the parameter of the function body with the expression, thus moving the computation into the called function.

Example:

Performing extract-parameter on the number 10

```
(foo 5 10)
(define foo (x new-var) (+ x new-vasr))
```

causes the code to change to

```
(foo 5)
(define foo (x) (+ x 10))
```

Rename function: Given a selected function-def node, the corresponding function definition (and all calls to it) are renamed. This transformation will prompt the programmer for the new name.

Move Into Interface: Given a function-def node, mark the function as part of the interface of the module being created. Moving a function definition into a module's scope also exports the function if it is referenced outside the module's scope.

Move out of interface (Remove): Given a function name in the list of functions in the module interface, move the corresponding function out of the interface list and place it back in the star diagram. Remove allows you to backtrack if you decide a given function shouldn't really be in the interface.

Things to remember:

- ERNIE only manipulates existing program structure. It doesn't add a completely new variable or function, but does allow you to create a new function based on a common expression or a new variable whose value is an existing computation. Think of ERNIE as letting you manipulate what gets to be an abstraction.

- ERNIE can only restructure the old version of the program – you can't save out a program, make changes to the program, then load the edited program back into ERNIE. However, if during editing, you find that you didn't restructure enough (or correctly), you can return to ERNIE and continue to restructure the restructured but unedited program.

- ERNIE is a research prototype. It doesn't always work well, and it can be hard to use. If you find yourself getting confused about what to do next, it's probably the tool's fault. Please bear with ERNIE and try using it for this task.

- ERNIE doesn't display comments. Any program loaded into ERNIE is stripped of comments. You can look at the file containing the original source code for the program to understand the program you're manipulating.

Appendix B

Source code for the Example Programs

B.A 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) that 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 ((wordno 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 shift2 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)))
((= i 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)
```

B.B Bussim Program

```
(define *route-distance* 0)

(define scenery
    (lambda (mileage)
      (let* ()
        (cond ((= mileage 0) (write "Bus Depot"))
      ((< mileage 10) (write "Road to Big City"))
      ((< mileage 20) (write "In Big City"))
      ((< mileage 30) (write "Road to Village"))
      ((= mileage 30) (write "In Village"))
      ((< mileage 40) (write "Road to Small City"))
      (else (write "In Small City"))))))

(define move-bus
    (lambda ()
      (let* ((i 0))
(write "Distance: ")
(write *route-distance*)
(scenery *route-distance*)
(set! *route-distance* (1+ *route-distance*))
(if (> *route-distance* 40) (set! *route-distance* 0))
t)))

(define simulation
    (lambda ()
      (let* ((i 0))
(set! *route-distance* 0)
(do ((i 0 (1+ i)))
    ((= i 100) nil)
  (move-bus)))))

(simulation)
```

Appendix 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 com- pletely. Clicks on all-words 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 all-words 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?	

time	action	spkr	dialogue	comments
	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.
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 number 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.	

Bibliography

[Adams 80] S. Adams. Pirate's adventure. *Byte*, 5(12):192–212, 1980.

[Aho 80] A. V. Aho. Pattern matching in strings. In R. V. Book, editor, *Formal Language Theory: Perspectives and Open Problems*, pages 325–347. Academic Press, New York, 1980.

[Aho et al. 86] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers, Principles, Techniques, and Tools*. Addison-Wesley, Reading, Mass., 1986.

[Atkinson et al. 95] D. C. Atkinson, W. G. Griswold, and C. McCurdy. Fast, flexible syntactic pattern matching and processing. In *18th International Conference on Software Engineering*, 1995.

[Ballance & Maccabe 92] R. A. Ballance and A. B. Maccabe. Program dependence graphs for the rest of us. Technical Report CS92-10, Department of Computer Science, University of New Mexico, 1992.

[Balzer et al. 76] R. Balzer, N. Goldman, and D. Wile. On the transformational implementation approach to programming. In *2nd International Conference on Software Engineering*, pages 337–344, 1976.

[Bentley et al. 92] R. Bentley, J. A. Hughes, D. Randall, T. Rodden, P. Sawyer, D. Shapiro, and I. Sommerville. Ethnographically-informed systems design for air traffic control. In *Computer-supported Collaborative Work '92*, pages 123–129, 1992.

[Bergland 81] G. D. Bergland. A guided tour of program design methodologies. *IEEE Computer*, 14(10):18–37, 1981.

[Berlin & Jeffries 92] L. Berlin and R. Jeffries. Consultants and apprentices: Observations about learning and collaborative problem solving. In *Computer-Supported Cooperative Work '92*, pages 130–137, 1992.

[Biggerstaff 89] T. J. Biggerstaff. Design recovery for maintenance and reuse. *IEEE Computer*, 22(7):36–49, 1989.

[Boehm 75] B. W. Boehm. The high cost of software. In E. Horowitz, editor, *Practical Strategies for Developing Large Software Systems*. Addison-Wesley, Reading, MA, 1975.

[Brooks 87] F. P. Brooks. No silver bullet: Essence and accidents of software engineering. *IEEE Computer*, 20(4):10–19, 1987.

[Bull 90] T. Bull. An introduction to the WSL program transformer. In *Conference on Software Maintenance 1990*, pages 242–250, 1990.

[Bush 85] E. Bush. The automatic restructuring of COBOL. In *Conference on Software Maintenance*, pages 35–41, 1985.

[Chen ss] M. Chen. A tool for planning restructuring of data abstractions in large systems. Master's thesis, University of California, San Diego, In Progress.

[Chi 94] M. T. H. Chi. Analyzing the content of verbal data to represent knowledge: A practical guide. *Submitted to Journal of Learning Sciences*, 1994.

[Cousin & Collofello 92] L. Cousin and J. S. Collofello. A task-based approach to improving the software maintenance process. In *Conference on Software Maintenance*, pages 118–126, 1992.

[Cousot & Cousot 77] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Fourth ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.

[de Balbine 75] G. de Balbine. Better manpower utilization using automatic restructuring. In *National Computer Conference*, 1975.

[Dumas & Parsons 95] J. Dumas and P. Parsons. Discovering the ways programmers think about new programming environments. *Communications of the ACM*, 38(6):45–56, 1995.

[Dybvig 87] R. K. Dybvig. *The Scheme Programming Language*. Prentice-Hall, Englewood Cliffs, NJ, 1987.

[Embley & Woodfield 88] D. W. Embley and S. N. Woodfield. Assessing the quality of abstract data types written in Ada. In *10th International Conference on Software Engineering*, pages 144–153, 1988.

[Ericsson & Simon 93] K. A. Ericsson and H. A. Simon. *Protocol Analysis: Verbal Reports as Data*. MIT Press, Cambridge, MA, 1993.

[Ferrante et al. 87] J. Ferrante, K. J. Ottstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, 1987.

[Flor & Hutchins 91] N. V. Flor and E. L. Hutchins. 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, editors, *Empirical Studies of Programmers: Fourth Workshop*, pages 36–64. Ablex, Norwood, NJ, 1991.

[Flor 94] N. Flor. *Dynamic Organization in Multi-Agent Distributed Cognitive Systems*. PhD dissertation, Cognitive Science Department, University of California, San Diego, 1994.

[Gray & Anderson 87] W. D. Gray and J. R. Anderson. Change episodes in coding: When and how do programmers change their code? In G. M. Olson, S. Sheppard, and E. Soloway, editors, *Empirical Studies of Programmers: Second Workshop*, pages 185–197. Ablex Publishing Corporation, Norwood, NJ, 1987.

[Griswold & Bowdidge 93] W. G. Griswold and R. W. Bowdidge. Program restructuring via design-level manipulation. In *Workshop on Studies of Software Design*, 1993.

[Griswold & Notkin 92] W. G. Griswold and D. Notkin. Computer-aided vs. manual program restructuring. *ACM SIGSOFT Software Engineering Notes*, 17(1), 1992.

[Griswold & Notkin 93] W. G. Griswold and D. Notkin. Automated assistance for program restructuring. *Transactions on Software Engineering and Methodology*, 2(3):228–269, 1993.

[Griswold 93] W. G. Griswold. Direct update of dataflow representations for a meaning-preserving program restructuring tool. In *ACM SIGSOFT '93: Symposium on the Foundations of Software Engineering*, pages 42–55, 1993.

[Harel et al. 90] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. Trakhtenbrot. Statemate: A working environment for the development of complex reactive systems. *IEEE Transactions on Software Engineering*, 16(4):403–414, 1990.

[Heath & Luff 92] C. Heath and P. Luff. Collaboration and control: Crisis management and multimedia technology in London Underground line control rooms. *Computer-supported Collaborative Work*, 1(1-2):69–94, 1992.

[Horwitz & Reps 92] S. Horwitz and T. Reps. The use of program dependence graphs in software engineering. In *14th International Conference on Software Engineering*, pages 392–411, 1992.

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

[Hutchins 90] E. Hutchins. The technology of team navigation. In J. Galegher, R. E. Kraut, and C. Egido, editors, *Intellectual teamwork: Social and technical bases of collaborative work.* Lawrence Erlbaum, Inc., Hillsdale, NJ, 1990.

[Hutchins et al. 85] E. L. Hutchins, J. D. Hollan, and D. A. Norman. Direct manipulation interfaces. *Human-Computer Interaction*, 1:311–338, 1985.

[Kay 95] J. S. Kay. *Path IDs: a mechanism for reducing software latency*. PhD dissertation, Computer Science and Engineering Department, University of California, San Diego, 1995.

[Kernighan & Ritchie 78] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice-Hall, Englewood Cliffs, New Jersey, 1978.

[Kernighan & Ritchie 86] B. W. Kernighan and D. M. Ritchie. The m4 macro processor. In *UNIX Programmer's Supplementary Documents, Volume 1 (PS1), 4.3 Berkeley Software Distribution, Virtual VAX-11 Version*. USENIX Association, Berkeley, CA, 1986.

[Kuck et al. 81] D. J. Kuck, R. H. Kuhn, B. Leasure, D. A. Padua, and M. Wolfe. Dependence graphs and compiler optimizations. In *8th Symposium on Principles of Programming Languages*, pages 207–218, 1981.

[Lampson 84] B. Lampson. Hints for computer system design. *IEEE Computer*, 1(1):11–28, 1984.

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

[Larus & Hilfinger 88] J. R. Larus and P. N. Hilfinger. Detecting conflicts between structure accesses. In *SIGPLAN '88 Conference on Programming Languages Design and Implementation (PLDI)*, pages 21–34, 1988.

[Leffler et al. 88] S. J. Leffler, M. K. McKusick, M. J. Karels, and J. S. Quarterman. *The design and implementation of the 4.3 BSD UNIX operating system*. Addison-Wesley, Reading, MA, 1988.

[Lehman & Belady 85] M. M. Lehman and L. A. Belady. *Program Evolution: Processes of software change.* Academic Press, London, 1985.

[Letovsky & Soloway 85] S. Letovsky and E. Soloway. Strategies for documenting delocalized plans. In *Conference on Software Maintenance*, pages 144–151, 1985.

[Letovsky 86] S. Letovsky. Cognitive processes in program comprehension. In *First Workshop on Empirical Studies of Programmers*, pages 58–79, 1986.

[Lientz & Swanson 80] B. Lientz and E. Swanson. *Software Maintenance Management: A Study of the Maintenance of Computer Application Software in 487 Data Processing Organizations*. Addison-Wesley, Reading MA, 1980.

[Liskov et al. 77] B. Liskov, A. Snyder, R. Atkinson, and C. Schaffert. Abstraction mechanisms in CLU. *Communications of the ACM*, 20(8):564–576, 1977.

[Miyake 86] N. Miyake. Constructive interaction and the iterative process of understanding. *Cognitive Science*, 10(2):151–177, 1986.

[Moen 90] S. Moen. Drawing dynamic trees. *IEEE Software*, pages 21–28, July 1990.

[Morgenthaler & Griswold 95] J. D. Morgenthaler and W. G. Griswold. Program analysis for practical program restructuring. In *Proceedings of the ICSE-17 Workshop on Program Transformations for Software Evolution*, 1995. Proceedings available as Tech. Report CS95-418, Department of Computer Science and Engineering, University of California, San Diego.

[Moriconi & Hare 85] M. Moriconi and D. F. Hare. Visualizing program designs through PegaSys. *IEEE Computer*, 18(8):72–85, 1985.

[Muller et al. 92] H. A. Muller, S. R. Tilley, M. A. Orgun, and B. D. Corrie. A reverse engineering environment based on spacial and visual software interconnection models. In *SIGSOFT '92: Fifth Symposium on Software Development Environments*, pages 88–98, 1992.

[NeXT Computer, Inc. 90] NeXT Computer, Inc. *NeXT Step Reference, Volume I*. Redwood City, CA, 1990.

[Norman 86] D. A. Norman. Cognitive engineering. In D. A. Norman and S. W. Draper, editors, *User Centered System Design*. Lawrence Erlbaum Associates, Publishers, Hillsdale, NJ, 1986.

[Ott & Thuss 89] L. M. Ott and J. J. Thuss. The relationship between slices and module cohesion. In *11th International Conference on Software Engineering*, pages 198–204, 1989.

[Parnas 72] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972.

[Parnas 79] D. L. Parnas. Designing software for ease of extension and contraction. *IEEE Transactions on Software Engineering*, SE-5(2):128–138, 1979.

[Quilici 93] A. Quilici. A hybrid approach to recognizing programming plans. In *Working Conference on Reverse Engineering*, pages 126–133, 1993.

[Reiss 87] S. P. Reiss. Working in the Garden environment for conceptual programming. *IEEE Software*, 4(6):16–27, 1987.

[Rich & Waters 88] C. Rich and R. C. Waters. The programmer's apprentice: A research overview. *IEEE Computer*, 21(11):10–25, 1988.

[Robins 85] G. Robins. Applications of the ISI Grapher. In *Fourth Annual Artificial Intelligence and Advanced Computer Technology Conference*, pages 105–129, 1985.

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

[Rowe et al. 91] L. A. Rowe, J. A. Konstan, B. C. Smith, S. Seitz, and C. Liu. The PICASSO application framework. In *14th ACM Symposium on User Interface Software and Technology*, pages 95–106, 1991.

[Schneiderman & Carroll 88] B. Schneiderman and J. M. Carroll. Ecological studies of professional programmers. *Communications of the ACM*, 31(11):1256–1258, 1988.

[Schneiderman 83] B. Schneiderman. Direct manipulation: A step beyond programming languages. *IEEE Computer*, pages 57–69, 1983.

[Schwanke 91] R. W. Schwanke. An intelligent tool for re-engineering software modularity. In *13th International Conference on Software Engineering*, pages 83–92, 1991.

[Sommerville 89] I. Sommerville. *Software Engineering*. Adsison-Wesley, Wokingham, England, 1989.

[Soni 91] D. Soni. Intelligent support for software maintenance. *Siemens Review*, 1991(Spring):14–18, 1991.

[Srinivasan 93] R. Srinivasan. *Automatic Software Design Recovery and Re-Modularization Using Source Transformation*. M.S. thesis, Queens University, Kingston, Ontario, Canada, 1993.

[Stallman 93] R. Stallman. *GNU Emacs*. Free Software Foundation, Cambridge, MA., November 1993.

[Stevens et al. 74] W. P. Stevens, G. J. Myers, and L. L. Constantine. Structured design. *IBM Systems Journal*, 13(2):115–139, 1974.

[Suchman & Trigg 91] L. A. Suchman and R. H. Trigg. Understanding practice: Video as a medium for reflection and design. In J. Greenbaum and M. Kyng, editors, *Design at Work: Cooperative Design of Computer Systems*, pages 65–89. Lawrence Erlbaum Associates, Hillsdale, New Jersey, 1991.

[Sutcliffe & Maiden 92] A. G. Sutcliffe and N. A. M. Maiden. Analysing the novice analyst: cognitive models in software engineering. *International Journal of Man-Machine Studies*, 36:719–740, 1992.

[Walz et al. 93] D. B. Walz, J. J. Elam, and B. Curtis. Inside a software design team: Knowledge acquisition, sharing, and integration. *Communications of the ACM*, 36(10):63–77, 1993.

[Weick 68] K. E. Weick. Systematic observational methods. In G. Lindzey and E. Aronson, editors, *The Handbook of Social Psychology*, pages 357–451. Addison-Wesley, Reading, MA, 1968.

[Weiser 81] M. Weiser. Program slicing. In *International Conference on Software Engineering*, pages 439–449, 1981.

[Weiser 84] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):35–357, 1984.

[Wildman 95] D. Wildman. Getting the most from paired-user testing. *ACM Interactions*, 2(3):21–27, 1995.

[Yang et al. 89] W. Yang, S. Horwitz, and T. Reps. Detecting program components with equivalent behaviors. Technical Report TR-840, University of Wisconsin, 1989.