Automated Support for Encapsulating Abstract Data Types*

Robert W. Bowdidge and William G. Griswold
Department of Computer Science and Engineering
University of California, San Diego
La Jolla, CA 92093-0114
{bowdidge,wgg}@cs.ucsd.edu

Abstract

A software engineer can use a meaning-preserving program restructuring tool during maintenance to change a program's structure to ease modification. One common restructuring action is to create a new abstract data type by encapsulating an existing data structure. Data encapsulation simplifies modification by isolating changes to the implementation and behavior of an abstract data type. To perform encapsulation, a programmer must understand how the data structure is used in the code, identify abstract operations performed on the data structure, and choose concrete expressions to be made into functions. We provide a manipulable program visualization, called the star diagram, that 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, 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 describe the design, implementation, and application of the star diagram, and evaluate its ability to assist data encapsulation in large programs.

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

To be presented at SIGSOFT '94 Symposium on Foundations of Software Engineering, December 6-9, 1994, New Orleans, LA.

1 Introduction

1.1 Background: Automated Assistance for Program Restructuring

An unanticipated modification to a software system is sometimes not easily integrated into the system's existing design, requiring changes to multiple modules [Parnas 72]. As repeated modifications are made, the system's design and implementation become increasingly less understandable until maintenance becomes unacceptably expensive. The only solutions are to reimplement the system or restructure it [Belady & Lehman 76]. Although restructuring is the more attractive approach, it is a complex, program-wide activity that is error-prone when performed manually [Griswold & Notkin 92].

Techniques have been described for a tool to automatically maintain a program's consistency as a tool user manipulates the structure of a program [Griswold 91][Griswold & Notkin 93], and a prototype tool for meaning-preserving restructuring of imperative Scheme programs [Dybvig 87] has been implemented. Such a tool improves the process of making maintenance changes by separating the process into two separate tasks. The first is to restructure the program—without changing its meaning—into a form that localizes the scope of the modification. The second task is to insert the modification into the restructured program. The meaning-preserving nature of the tool makes testing the restructured program for added errors unnecessary. Localizing the scope of the modification makes the change easier to perform and reduces the chance of introducing errors into the program.

Our original implementation of this tool supported restructuring actions by presenting the text of the program in a window, allowing the tool user to select a portion of a program and apply a restructuring operation on it and related components. These operations include extracting a new function from inlined statements, inlining functions, extracting embedded constants or expressions as parameters to functions, and the like. When a transformation is applied, the tool checks that it will be meaning-preserving. If the check fails, the tool does not perform the change, but instead reports the nature of the problem. Interprocedural data flow representations with alias information are used to perform these checks. Although such representations are expensive to compute, the cost is largely eliminated by incrementally updating the representations after each transformation [Griswold 93]. Transformations similar to ours have been developed for object-oriented systems [Opdyke & Johnson 90][Opdyke 92][Johnson & Opdyke 93], but automatically preserving meaning is more difficult in this domain.

1.2 Problem: Performing Data Encapsulation

Data encapsulation [Parnas 72][Liskov et al. 77] is a useful restructuring activity because hiding the code related to an evolving data structure limits the extent of changes to the scope of a module. Typically, encapsulating a data structure requires hiding the computations accessing a global data structure behind an intuitive, orthogonal set of operations.

In spite of our tool's support for global, meaning-preserving transformations, we have found that the restructuring task of encapsulating a data structure is not easy to perform using a textual presentation of the source code. The source code provides a significant amount of information unrelated to the computations on the data structure, obfuscating the data structure's relationships to the rest of the program. In addition, references to the data structure are scattered throughout the program, often in multiple files, making it difficult to identify all of the information that is required to choose appropriate interface operations. Without additional support, a programmer must methodically search to identify all computations on the data structure and keep careful records, or otherwise risk performing an ill-planned encapsulation.

In particular, the scattering of references can encourage a programmer to make interface design decisions locally. This tendency can result in defining operations early-on that are at a different "level of abstraction" than others discovered later. It is costly to go back and correct these conflicting interface decisions, because the affected code is distributed throughout the system. Similarly, the scattering of references will make it difficult to see the similarities among computations on the data structure. Such similarities are important, because noting repeated computations can suggest to a programmer that they represent an abstract operation. Even 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.

Any computation that references a data structure that is being encapsulated is a possible candidate for inclusion in the new module. A text searching tool like `grep` [Aho 80] provides assistance in locating direct references, but its lexical and line-based orientation yields false matches, includes irrelevant information, misses surrounding context, and misses cases such as indirect reference through assignment. Even if these issues are resolved, simply listing all the computations that reference the data structure potentially involves a substantial amount of text. Also, such a solution does not show the similarities among computations, nor assist in the actual manipulation.

After making these observations, we designed and implemented a direct-manipulation graphical restructuring interface [Griswold & Bowdidge 93] based on structure charts [Stevens et al. 74] (See Figure 1). Such a view reveals the high-level structure of the program and allows the tool user to directly change the structure through manipulations on the view. This interface is helpful in performing restructurings relevant to functional decompositions of programs, but does not support data encapsulation. In particular, the structure diagram interface is too high-level, only showing function definitions, calls, and global variable references. There is no natural access to the computation around a global variable reference to a data structure.

The remainder of this paper describes our solution to this problem, called the *star diagram*. We define the star diagram, provide an example of its use, discuss aspects of the approach not captured in the example, sketch the implementation, describe some related work, and conclude with a summary of our experience with graphical manipulation of programs.

2 Graphical Assistance for Encapsulation: The Star Diagram

The problems we encountered in restructuring with the text view and the structure diagram led us to a graph-oriented, direct-manipulation approach, called the *star diagram*. It elides unnecessary information, brings all computations on the data structure together, and shows overlaps of similar computations (See Figure 2; a portion of the actual code displayed is shown in Figure 1). To a first approximation, the star diagram is like a data flow graph—with nodes as operations and edges as data flows between them—but showing only the data flows generated by references to the data structure being encapsulated. In the case of the star diagram, however, all references to the variable are denoted by a single node, which results in all the relevant computations on a data structure appearing as a tree of computations emanating from the variable node:

- 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 represented as a 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

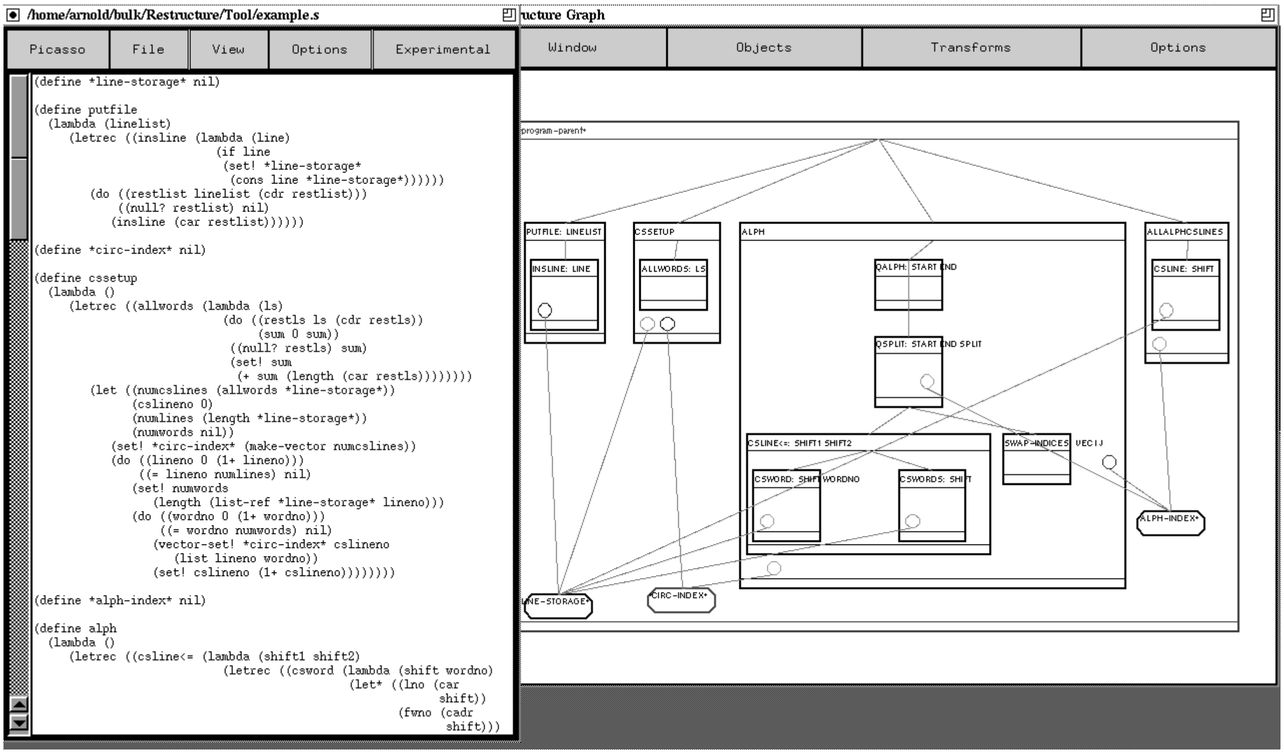


Figure 1: Text view and structure diagram in the restructuring tool.

(represented as shading appearing behind nodes) is what gives the star diagram its tree shape.

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

The resulting tree, then, 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 the list `*line-storage*` (Figure 2), the path `*line-storage*` → `list-ref` → `length` denotes the Scheme expression `(length (list-ref *line-storage* lineno))`, which computes the number of words in a list of lines of text. Hence, 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 could 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 2, the `length` path is actually a path of stacked nodes whose corresponding expressions come from the functions `cssetup`, `csword`, and `csline`.

In order to provide additional context and improve the selectability of computations, there are two key departures from a data flow graph. In particular:

- 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. A few are listed here. Function call nodes have thick borders to help identify where related computations may be hidden, perhaps leading to an inlining

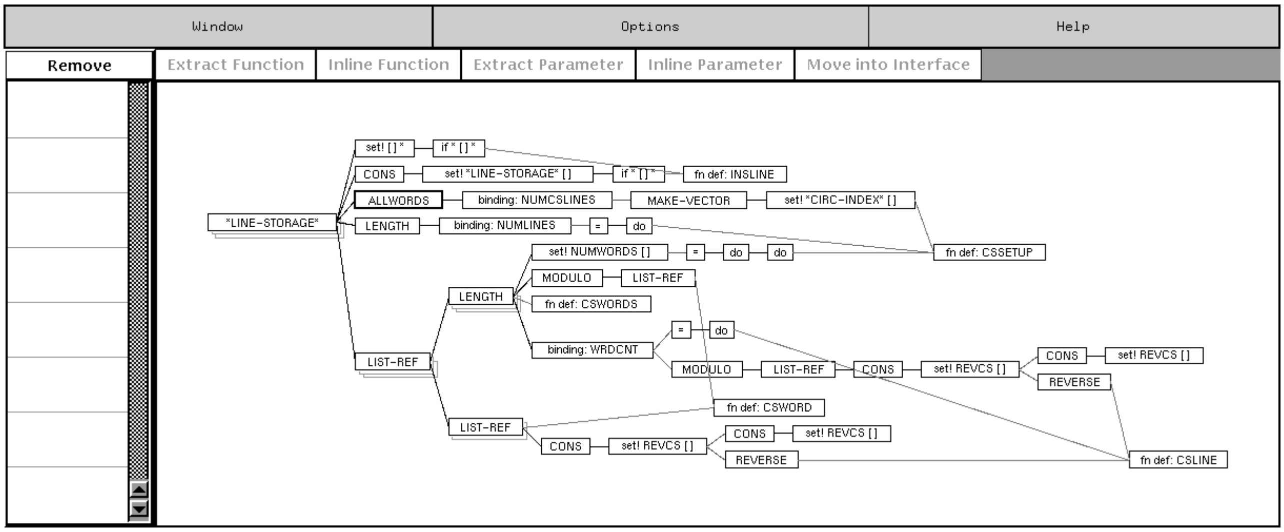


Figure 2: Star diagram for the array `*line-storage*` in the KWIC program. In the left column, the scrolling list displays the current functions in the interface, and the Remove button at the top allows the tool user to remove a function from the interface so it can be manipulated using the star diagram. The main window consists of the five transformation buttons at the top, and the star diagram in the rest of the window space.

---

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. To disambiguate the effects of nodes, built-in operations such as `set!` and `if` indicate the parameter position of the edge with square brackets. This feature could be extended to user-defined function calls. We are also experimenting with other customizations to expose additional relationships.

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. To apply a restructuring transformation to the program via the star diagram, the tool user clicks on a node, then clicks on a button for the desired restructuring transformation. The chosen transformation is immediately checked against the program to ensure it can preserve meaning, and if so, it is then performed on the text and the star diagram is updated to reflect the changes.

In order to provide easy access to other views on the program, double-clicking on a node causes the program text corresponding to the node to be displayed and highlighted in the program text view. When multiple statements are associated with a node, the first is shown immediately; the tool user can scroll the text view to see the rest.

We have found that there are six basic restructuring transformations that are performed when encapsulating from the text view of the restructuring tool. In the star diagram, these transformations are specifically tailored to the task of encapsulation to make them easier to use.

- 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. 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 before extracting them.

- Inline Function. Inlining a function call replaces the call with the body of the function. Inlining is the opposite of creating 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.

- 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.

- 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.

  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 so that the function body now encapsulates it.

- Move into Interface. Moving a function definition into a module's scope also exports the function if it is referenced outside the module's scope.

  In the star diagram, when this transformation is applied to a function definition node at the right of the diagram, the function node and all of its nodes on the paths back to the root are removed. 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.

- Move out of Interface. Of course, it may also be necessary to move a function out of an interface. 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.

3 Performing Data Encapsulation with 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.

To gather evidence on these issues, we have restructured two programs with the star diagram, Key Word In Context (KWIC) [Parnas 72] and a transportation simulation [Griswold & Notkin 93]. Here we briefly describe a simple encapsulation of a global data structure in a functional decomposition of KWIC. The KWIC indexing program takes a text file as input and produces the list of words appearing in the document, with the context of each word appearing with it (i.e., the rest of its enclosing line, circularly shifted to place the keyed word at the beginning of the line). In his analysis of KWIC, Parnas suggests an enhancement to allow storing portions of the data in secondary storage. Unfortunately, in the functional decomposition, the list of lines is directly accessed by every module of the program. To facilitate the enhancement, Parnas suggests hiding the direct references to the line list inside a new Line Storage module. Once this encapsulation is performed, it is possible to change the implementation of the module to accommodate secondary storage of lines without affecting the interface of the module or the implementations of other modules in the system.

This process is a classic data encapsulation, which should be performable using the star diagram. Our Scheme version of KWIC consists of about 200 lines of code. The data structure Parnas encapsulated is called `*line-storage*` in our version.

Extracting two line-oriented operations. To begin encapsulating the list, the tool user creates a star diagram for the list `*line-storage*`. Figure 2 shows the star diagram in the midst of restructuring KWIC. (Three transformations—all Inline Parameter—were performed before this point to shorten the example.) 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`. 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 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 is tempted to extract 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 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, 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

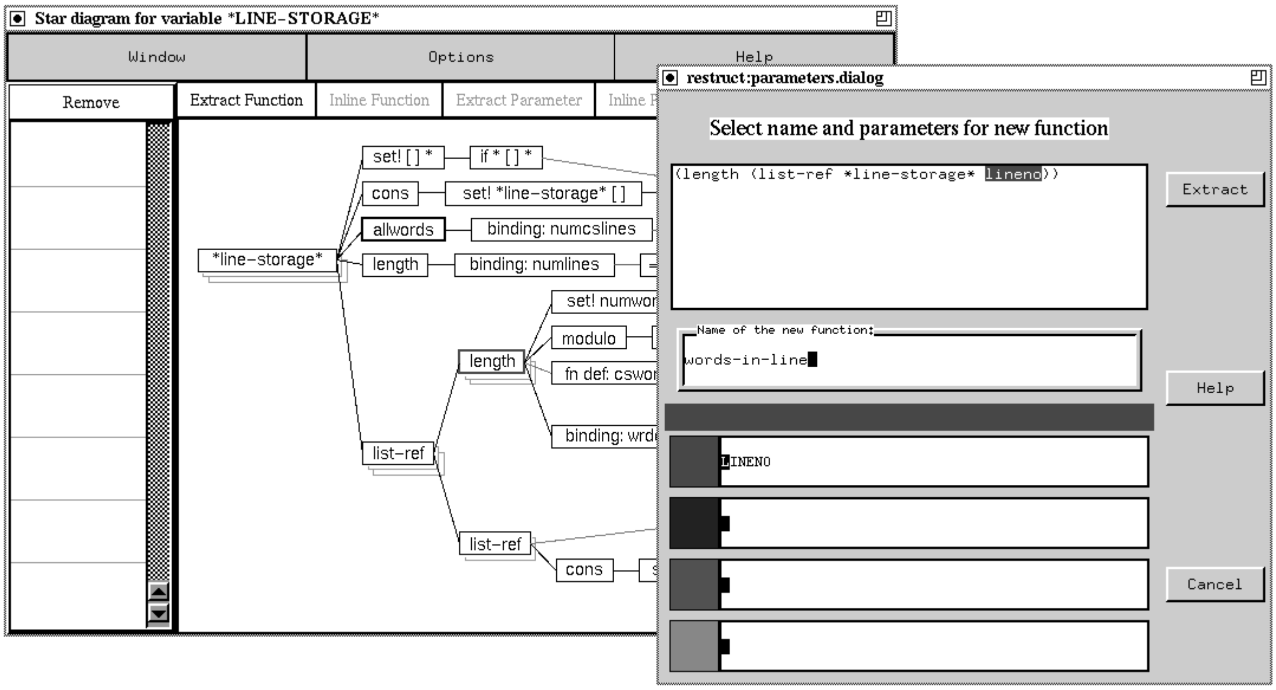


Figure 3: After selecting the length node in the path `*line-storage*` → `list-ref` → `length` and specifying that all four occurrences should be extracted as a new function, the tool prompts for the new function name and the parameters for the new function (as seen above). Once the parameters and function name are specified, the transformation is performed.

to be made parameters to the new function (See Figure 3). In this case, there is one parameter, the variable `lineno` identifying the line of interest. The user fills in these values and a name for the parameter, and the tool user clicks on the Extract button in the dialog box. At this point the underlying 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 since they now reference `*line-storage*` through function calls. Since the tool user wants this new function to be part of the module interface, the `words-in-line` definition node is selected and the Mark as Interface operation is invoked, removing the operation and its path from the star diagram and placing its name on the interface list.

A similar sequence is performed for extracting `get-word-in-line`. Figure 4 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 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 since 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 5 shows the star diagram after this transformation. Now `allwords` hides a use of `*line-storage*`, so the tool user marks `allwords` as a member of the 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

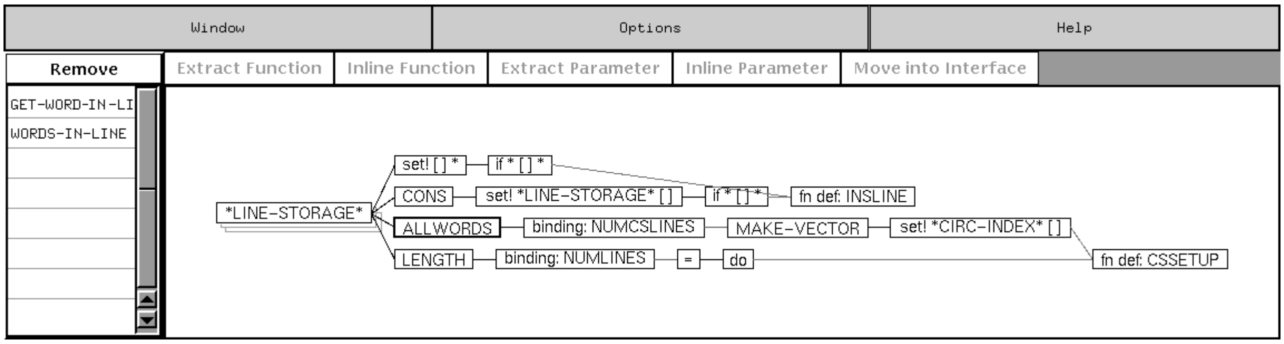


Figure 4: Star diagram after the functions `get-word-in-line` and `words-in-line` have been extracted.

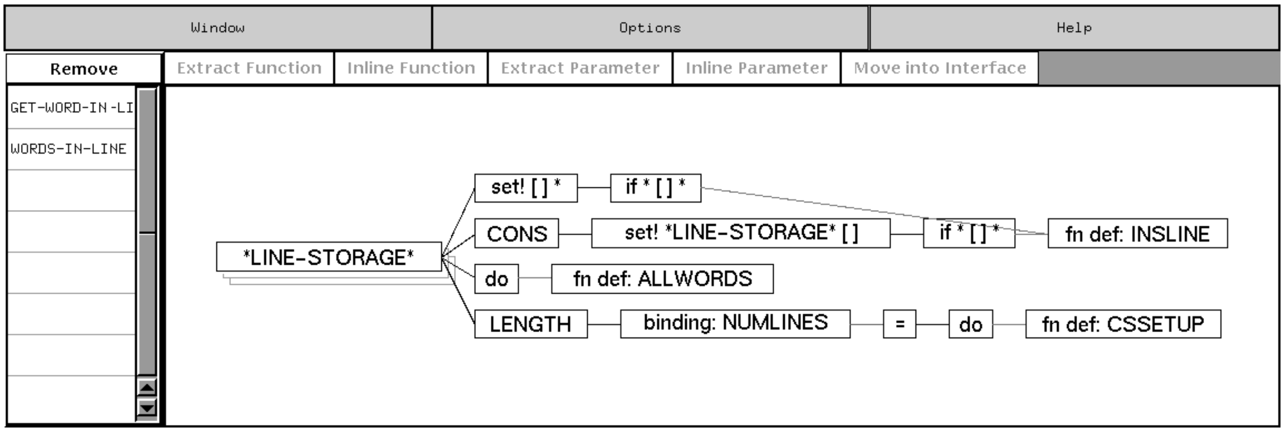


Figure 5: Star diagram after Inline Parameter has been applied to `allwords`, but before function is placed in the interface.

`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 it belongs in the interface of the Line Storage module, and marks it as part of the 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—is confined to the Line Storage module.

4 Discussion

The example above demonstrates that the star diagram can be used for understanding as well as performing the encapsulation. In particular:

- The tool user was able to see all of the uses of the data structure in a single, condensed view.
- Contextual information in the diagram helped identify properties that drove the design of the abstract interface. Stacks highlighted frequently occurring computations that identified potential abstract functions. Also, references to the data structure by a function call from a different level than the other operations in the diagram (`allwords`) indicated an inconsistency in the existing interface. The names of the objects in the star diagram provided additional informal knowledge about the program.
- All the needed transformations could be applied through the star diagram.
- 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 be inferred based on the module being created. In the text-based tool, all such decisions are made explicit to make the tool's actions as general as possible.

We have also observed 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.

A few issues could not be conveyed by our simple example. We discuss these in the following subsections.

4.1 Customizing Selections and Selecting Parameters during Function Extraction

One issue not adequately addressed in our example is the customization of selections and the selection of parameters during function extraction. 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, it is hard to display and manipulate all the selections simultaneously to properly identify the appropriate extension.

Parameter selection presents a similar problem. The textual view shown in Figure 3 uses an example textual 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. Since the different components cannot be integrated into a single function body, they are assumed to be outside the function body—their results are passed as arguments into the function. However, sometimes the tool user wants a similar component to be passed as an argument as well. This requires customizing the default parameter selection made by the tool, which sometimes requires looking at all of the selections to make the best choice, becoming unwieldy.

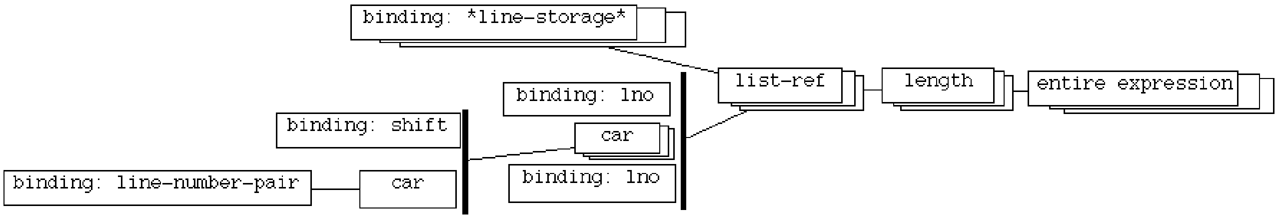
An alternative approach is to extend the star diagram concept. Following our initial analogy between a star diagram and a data flow graph, the basic star diagram can be thought of as a graphical display of the forward slices [Weiser 84] of every use of a variable.[1] Similar prefixes of the slices are then overlaid to identify common expressions. This analogy naturally leads to the question of whether there is an analogy to backward slicing in the star diagram. This question led to the idea of extending the star diagram to display backwards from nodes in the star diagram to show paths other than those related to the data structure being encapsulated. We found that, in general, this extension displays too much information and it is not particularly useful for selecting abstract operations.

However, once a path is selected 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 6). 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 denotes nesting or data flow through a variable assignment, although edges due assignments have been omitted from the Figure. 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 selected for encapsulation. 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 selections, only the paths common to the extended selection are retained for extraction. For example, in Figure 6, the basic selection includes only the top path, since 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,

[1] A forward slice is the set of expressions encountered by computing the transitive closure of the forward data flow and control dependence relationships from a variable. Semantically speaking, a forward slice contains all those computations whose results may be influenced by the value of the variable during program execution. Likewise, a backward slice is computed by following data flow and control dependence relationships backwards. A backward slice of a variable contains all those computations that may influence its value.




```
(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 6: 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.

---

the computations excluded must become arguments to the function so that their values can reach the included code.

4.2 Customizing Selections

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. 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 investigated a few ways of including computations not related by data flow or nesting in the star diagram:

1. `let` nodes could be included in the diagram, so that the tool user is aware of extra levels of nesting and can select them. However, including `let` nodes resulted in cluttered diagrams with some programming styles, and scoping is an imperfect indicator of relatedness.

2. Rather than using the backward star diagram to customize the selection and choose parameters, the tool user could customize 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 constructed a text-like view that shows the 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.

4.3 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 the line storage, but a different implementation might support several such stores.

Although hiding types rather than individual data structures is somewhat complicated 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 (See Section 5). 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 references to the instances are collapsed to a single root node, as in the diagram for Scheme.

The potential shortcoming of this approach is that two data structures of the same representation type, particularly two arrays, might be used for sufficiently different purposes that they are not really instances of the same type abstraction. One way to avoid such false matches is to allow the tool user to selectively include and exclude instances of the type.

Consequently, the tool user can perceive how the diagram is affected and decide what instances to keep.

Type-oriented encapsulation also results in potentially more informative diagrams. Consider the `allwords` call from the example in Section 2. 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. However, it might have been more helpful to actually see the body as part of the star diagram, since this would reveal overlaps with other computations. With type-oriented encapsulation, both the call and the body are in the diagram, since the parameter is of the type of `*line-storage*`. One potential downside is that displaying both might cause scalability problems (See Section 4.6). One way to control this feature, however, is to show parameter paths only if the tool user requests it. We are currently investigating the potential impacts of these alternatives.

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, the casts would appear on the star diagram when the object of the examined type gets cast to the new type. The tool user, upon noting the cast, could examine the source code to understand why the cast occurs (and restructure, if necessary, from that view). Alternatively, since the cast and the operations upon the recast value are visible in the star diagram, the tool user could manipulate the expression containing the cast in that view as well.

4.4 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 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.[2] 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 could 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.

4.5 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 one or 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 which 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.

4.6 Scalability of the Star Diagram

For best results, the most frequently accessed parts of a star diagram need to be viewable as a single unit, fitting on a single screen. Since restructuring to encapsulate is most useful for large programs, scalability of the star diagram is essential. We hypothesize that the star diagram is naturally scalable, because there are only so many unique tasks that can be performed on a single data structure. With stacking, then, the number of interesting paths in the graph will be bound by this number of unique tasks.

To verify our hypothesis that the star diagram is scalable, we produced C star diagrams for larger programs. The C diagramming tool is similar to the Scheme diagramming tool, but the uses of temporary variables that interrupt calculations were inferred manually.

We processed two C programs and generated the star diagram for some of their global data structures. The Adventure Interpreter program is a small C program; the data structures chosen have a small (`currentRoom`) and large (`items`) number of references. The Emacs program is the C source code for the GNU Emacs editor; one variable (`minibuf_window`) represents a small data structure with few encapsulation choices, and the other (`current_buffer`) represents a large data structure with 45 members and a very large set of possible operations. As shown in Table 1, when their star diagrams are compared to KWIC's, the number of paths and their lengths increases slowly with respect to the total size of the program. In fact, all the star diagrams except for `current_buffer` fit on a

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

single page. Although the greps are smaller (we have omitted them and the star diagrams due to space considerations), they do not provide as much context and it is not possible to view all similar instances together *and* in context.

On the other hand, we found that the number of paths in a star diagram—which is equal to the number of leaves—tends to grow in proportion to the number of references to the data structure. If the number of references to a variable is in proportion to the size of the program and redundancy is not widespread, then the star diagrams can become quite bushy at the leaves.

However, we have observed that users of the star diagram tend to begin at the left of the diagram looking for appropriate abstractions, and they tend to be found where stacking occurs (unless the data structure is already encapsulated), before much branching has occurred. On the other hand, function definition nodes at the far right of the diagram are also frequently used to discern context. To leverage this style of usage, paths extending beyond the window's right border can be elided so as to include the needed function definition nodes at the edge. When the diagram is pulled to the left, elided nodes are added to the right of the diagram. Because paths closer to the root exhibit much more stacking, the tree is less bushy there and easier to manage. For example, Emacs's `current-buffer` has an initial branching of 60. But the next level branches to only 301 paths, and that level appears to contain most of the functionality that is worth encapsulating. (If not, then the resulting module could contain an enormous number of operations!) To work at this level, then, requires an acceptable amount of vertical scrolling to examine and manipulate all the paths. In contrast, there is no way to use the `grep` information in such a compacted form. In particular, the `grep` output is arranged by file and line number, not by a hierarchy of unique operations applied to the data structure.

Alternatively, structures with a large number of fields (such as the `current-buffer` structure in Emacs) might possibly be restructured in sections, if the sections are sufficiently independent. For example, it might be feasible to first create the functions for manipulating the entire structure, and then encapsulate the fields with operations.

4.7 Showing Calling Context

The inverse problem to scalability is showing too little information. In particular, consider a function definition that is "too small", so that there is significant coupling to its callers with respect to the data structure being encapsulated. Inlining may be required to bring the body into the context of the callers. However, 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 primary concerns are how to display this extra information in the space provided and where to terminate these extensions. However, since we observed that the diagram is used left-to-right, the natural approach is to extend the diagram as necessary, probably at the direction of the tool user.

5 Constructing the Star Diagram

The construction and manipulation of the star diagram depends on the underlying restructuring tool and graphical interface support. The restructuring tool preserves meaning and computes key semantic relationships by referring to a program dependence graph (PDG) [Ferrante et al. 87] constructed from an abstract syntax tree (AST) of the program. The PDG implementation is taken from the Curare system [Larus 89][Larus & Hilfinger 88]. To improve the speed of transformations, incremental updates of the PDG have been designed and implemented [Griswold 93]. These components consist of about 38,000 lines of Common Lisp. The direct manipulation subsystem is implemented with the PICASSO user interface toolkit [Rowe et al. 91] and ISI Grapher [Robins 85], consisting of about 51,000 lines of Common Lisp. The code for generating and manipulating the star diagram is about 2,000 lines.

The star diagram is generated from the AST and PDG, using nesting of expressions in the AST for most connections, but using the PDG where syntactic nesting is broken up by assignments to temporary variables. A star diagram is generated roughly as follows:

1. Search the AST for identifier nodes corresponding to the variable to be encapsulated.
2. For each identifier node, recursively move up the AST, collecting the nodes along the path.
3. However, if a node on a path is an assignment to a variable local to the function, ignore the syntactic parent and instead follow all the data flow edges from the assignment to the uses of the assigned variable. Continue searching up from these uses.
4. Merge prefixes of paths that have matching node contents.
5. Connect each leaf to a node representing its containing function definition.
6. Draw the resulting graph.

To produce a less-cluttered view, the actual construction algorithm skips over certain nodes that serve little purpose for encapsulation, such as scope constructs. The construction algorithm for the C star diagram is similar. Both are fast enough

program	LOC	variable	var. refs.	grep LOC	nodes	distinct paths	length max.	length avg.	nodes unmerged
KWIC	200	`*line-storage*`	10	10	42	13	10	6.5	72
Adventure Interpreter	1700	`currentRoom`	36	34	86	27	8	5.6	172
		`items`	83	78	130	60	15	7.8	601
Emacs	76000	`minibuf_window`	57	65	137	48	12	5.6	262
		`current_buffer`	1909	1354	3634	1605	26	9.5	16408

Table 1: Measurements of scaling properties of star diagrams.

that the star diagram's construction time is dominated by the graph layout time.

Transforming the AST via the star diagram is not difficult, since there is a straightforward mapping from the star diagram nodes to (sets of) AST nodes. The primary challenge is extracting a path that contains an assignment to a temporary, because the computation of the temporary may not be contiguous with the rest of the computation. In this case the computation of the assignment must be moved next to the rest of the computation, which sometimes requires introducing an extra local scope.

6 Related Work

6.1 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 imply 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 types 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 suggesting the weaker communicational cohesion might imply that 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 4.2).

6.2 Fully Automated Data Encapsulation

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.

7 Conclusion

Tool-assisted program restructuring can improve the structure of a program in order to lower the excessive costs of maintenance. However, a text-based restructuring tool sometimes presents information inappropriate for the task and obscures information that is required. With the star diagram, the software engineer can work directly with the concepts and details necessary for data encapsulation. By eliding unnecessary details and displaying only the computations that can be manipulated, the star diagram reduces the amount of information displayed and allows the tool user to focus on the task at hand. Overlaying similar expressions further compacts the information in the view, reveals redundancy that can guide abstraction decisions, and assists encapsulation by allowing a set of similar computations to be manipulated together. Manipulating the source code through the star diagram allows the tool user to work at a natural conceptual level and to avoid switching context. Also, transforming the source along with the star diagram ensures that the encapsulation is feasible at each step.

In the process of designing the structure diagram, and then the star diagram, we discovered three principles that guided the star diagram's design and contributed to its success. The success of these principles in our views suggests that they may be useful in the design of similar tools.

- *Limit the problem domain.* By providing an elided and abstracted view of the program specialized to a particular task, we can provide intellectual leverage for that task. Trying to support more general manipulations requires visualizing more components and dependences, perhaps resulting in a cluttered view.

- *Views should be derivable from source code.* If the view is not derived from the source code or vice versa, then there is a possibility of the two becoming inconsistent. In particular, an independent view may require a tool user to manually specify links between the source code and the view when either is changed outside the context of the tool.

- *Manipulations must have predictable effect.* Since a programmer is expected to manipulate the source code at a later time, manipulations of a view must have predictable effects in the code. In our tool, the predictability requirement is met primarily by the meaning-preserving nature of the changes, guaranteeing that although the program's form may change, the program will behave the same. Also, we have attempted to always perform the smallest change possible to achieve the tool user's desires. Making changes that the tool user does not require might be changes that are unwanted.

The effectiveness of these design principles and the star diagram has been tested to some extent through informal experiments. The star diagram will be studied and refined further through controlled case studies of programmers performing encapsulation with the star diagram. Further development and use of the C star diagramming tool will lead to a better understanding of restructuring in large systems.

The design and application of the star diagram demonstrates that program manipulation can be performed through a view other than the source code. The success of this view derives from its focused task, appropriate visualization, and direct connection to the source code, which are enabled by the underlying meaning-preserving restructuring tool and the ability to navigate to other views when necessary. Abstracting away unnecessary details and presenting information obfuscated in the source code supports thinking primarily in terms of the program's design.

Acknowledgments

We thank David Morgenthaler for the C abstract syntax tree generator used to create C star diagrams. We also thank Darren Atkinson, Jonathan Kay, and David Noelle for their feedback on the star diagram, and Daniel Jackson, Jennifer Schopf, and the referees for their constructive comments on this paper.

References

[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.

[Belady & Lehman 76] L. A. Belady and M. M. Lehman. A model of large program development. *IBM Systems Journal*, 15(3):225–252, 1976. Reprinted in M. M. Lehman, L. A. Belady, editors, *Program Evolution: Processes of Software Change*, Ch. 8, APIC Studies in Data Processing No. 27. Academic Press, London, 1985.

[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 abtract data types written in Ada. In *Proceedings of the 10th International Conference on Software Engineering*, pages 144–153, April 1988.

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

[Griswold & Bowdidge 93] W. G. Griswold and R. W. Bowdidge. Program restructuring via design-level manipulation. In *Proceedings of the Workshop on Studies of Software Design*, Baltimore MD, May 1993. To appear as a Springer-Verlag Lecture Notes in Computer Science. Proceedings available as Queen's University Department of Computing and Information Science, External TR No. ISSN-0836-0227-93-352.

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

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

[Griswold 91] W. G. Griswold. *Program Restructuring to Aid Software Maintenance*. PhD dissertation, University of Washington, Dept. of Computer Science & Engineering, August 1991. Technical Report No. 91-08-04.

[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, December 1993.

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

[Johnson & Opdyke 93] R. E. Johnson and W. F. Opdyke. Refactoring and Aggregation. In *Object Technologies for Advanced Software*, volume 742 of *Lecture Notes in Computer Science*, pages 264–278. First JSSST International Symposium, November 1993.

[Larus & Hilfinger 88] J. R. Larus and P. N. Hilfinger. Detecting conflicts between structure accesses. In *Proceedings of the SIGPLAN '88 Conference on Programming Languages Design and Implementation*, June 1988. *SIGPLAN Notices*, 23(7).

[Larus 89] J. R. Larus. *Restructuring Symbolic Programs for Concurrent Execution on Multiprocessors*. PhD dissertation, UC Berkeley Computer Science, May 1989. Also Technical Report No. UCB/CSD 89/502.

[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, August 1977.

[Opdyke & Johnson 90] W. F. Opdyke and R. E. Johnson. Refactoring: An aid in designing application frameworks and evolving object-oriented systems. In *Proceedings of the 1990 Symposium on Object-Oriented Programming Emphasizing Practical Applications*, pages 274–282, September 1990.

[Opdyke 92] W. F. Opdyke. *Refactoring: A Program Restructuring Aid in Designing Object-Oriented Applications Frameworks*. PhD dissertation, University of Illinois at Urbana-Champaign, Dept. of Computer Science, 1992. Technical Report No. 1759.

[Ott & Thuss 89] L. M. Ott and J. J. Thuss. The relationship between slices and module cohesion. In *Proceedings of the 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, December 1972.

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

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

[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 *Proceedings of the 13th International Conference on Software Engineering*, pages 83–92, May 1991.

[Srinivasan 93] R. Srinivasan. Automatic software design recovery and re-modularization using source transformation. M.s. thesis, Queens University, Kingston, Ontario, Canada, 1993.

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

[Weiser 84] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, July 1984.