Program Restructuring via Design-Level Manipulation

William G. Griswold and Robert W. Bowdidge

Department of Computer Science & Engineering
University of California, San Diego
La Jolla, CA 92093-0114, U.S.A.
{wgg,bowdidge}@cs.ucsd.edu

Abstract. A meaning-preserving program restructuring tool can be used by a software engineer to change a program's structure to better support modifications during maintenance. Our implementation of such a tool performs restructuring transformations on code fragments that are selected using a text-based interface. However, a text representation does not represent program structure well because some component relationships that we conceptualize as structure are not readily observable in the program text. For example, structural properties such as module uses or procedure calls are represented by references to names rather than by proximity or direct linkage. Since restructuring is primarily a design-oriented activity, this design information must be readily available. Although the program text may be the wrong representation to manipulate, we still want transformations to directly affect the implementation. To solve both these problems, we use a graphical representation of the program design that displays the program's structure and, unlike visualization tools, permits direct manipulation of the structure to perform transformations on the implementation. We introduce a direct manipulation graphical interface that meets these criteria. We describe its design and implementation, and discuss the special problems of direct-manipulation program restructuring at the design level, and relate it to other approaches for manipulating designs.

1 The Problem

Software is perceived to be too expensive relative to its quality. Since maintenance is the dominant phase in the program life-cycle [9], substantially reducing the cost of software requires lowering the cost of maintenance.

As useful software ages, modifications to meet the needs and demands of users are layered upon the original implementation. Modifications unanticipated in the design sometimes are not easily integrated into the existing implementation, requiring changes to multiple modules in the system to complete a single change [12]. As repeated modifications are made, the design and implementation become increasingly less understandable until maintenance becomes unacceptably

* This work was supported in part by NSF Grant CCR-9211002.

expensive, and the only solutions are to reimplement the system or restructure it [8].

Restructuring is the more attractive approach. However, it is a complex, program-wide activity that is error-prone when performed manually [5]. Techniques have been described for automatically maintaining the program's consistency as the user manipulates the structure of a program, and a prototype tool for performing meaning-preserving restructuring has been implemented using the techniques [4] [6]. The tool improves the process of making maintenance changes by separating the process into two sequential, independent tasks. The first is to restructure the program, without changing its meaning, into a form that localizes the scope of the modification or otherwise improves the structure to prepare for the modification. The second task is to insert the modification into the restructured program. The meaning-preserving nature of the tool obviates the need to test the restructured program for added errors before proceeding with the modification.

In the prototype tool, which restructures programs written in the imperative programming language Scheme [2], the software engineer manipulates the structure of the program by selecting expressions in the source code as parameters to a restructuring operation (Fig. 1). The primary benefit of restructuring source code is that it is the most familiar and commonly available representation [17]. Also, restructuring the source does not require changing representations between restructuring and modification.

However, manipulating a program's structure through its implementation is inappropriate because manipulating structure is a design issue, not an implementation issue. In particular, the program text contains information that is irrelevant to the manipulation of the structure, and presents the program at a level of detail ill-suited to support modification of the program structure. Also, the software engineer cannot immediately perceive the structure of the program in the program text—it must be carefully inferred. The relation of a variable use or function call to its definition must be identified through references to an address or symbol, rather than through proximity or visible links. Moreover, the implementation displays details that are unrelated to the structure of the system, such as syntax added for readability or the sequence of operations performed by a procedure. For these reasons, restructuring a program by manipulating textual objects is awkward. A representation that better represents structure and design information is needed.

2 Graphical Interfaces for Restructuring

One possible solution is to use a graphical design notation to represent the program. Graphical representations of design are used in many existing tools [10] [13] because the pictorial representation of structure often can convey the needed information in a more readily assimilable manner. A graphical representation can convey program component relationships through direct linkage and grouping, allowing immediate comprehension. A graphical notation also allows the soft-

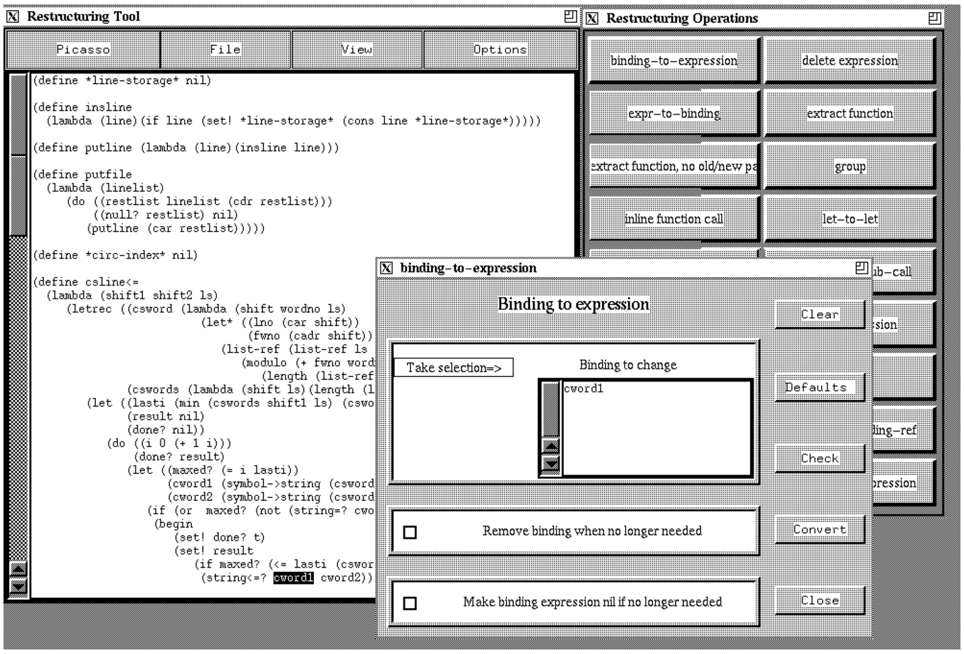

Fig. 1. Text-oriented user interface for Restructuring Tool. The left window contains the program text, and allows selection of expressions within the code. The upper right window contains a button for each possible transformation. When the transformation button is pressed, a panel such as the lower right window appears. To perform a transformation, the user fills in the parameters of the transformation through selection of program text or direct input, then presses the Convert button. When the transformation occurs, the program text window is updated to contain the transformed program.

ware engineer more freedom to choose the relations to be presented, eliding the implementation details that are deemed unnecessary to restructuring, and reducing the number of elements a programmer must examine when performing restructuring operations. A suitable notation can represent objects in a structure familiar to the software engineer. A rich notation also allows information not normally visible to be displayed on top of the structure. For example, variable use between modules might be discouraged in a given methodology; the notation could highlight these instances of high coupling.

Finally, a well-chosen notation can go beyond traditional visualization to allow restructuring transformations to be performed upon the objects underlying the notation. In order for the manipulation of the design notation to cause transformations upon the implementation text, mappings must exist between objects in the notation and the code in the implementation, and transformations on objects in the notation must have mappings to transformations on the implementation. Given these mappings, the feasibility of a restructuring and its precise impacts on the implementation can be known immediately.

3 A Notation for Graphical Restructuring

We have implemented an interface for the restructuring tool that presents a manipulable graphical representation of the program structure.

The graphical notation we chose (see Figs. 2 and 4) is similar to the structure chart notation described in “Structured Design” [16]. We use the notation because it represents many of the objects we intend the user to manipulate, and presents relations necessary for understanding how to restructure a program. The original notation presents functions, call-graph information, and intra-function variable references. Our extensions include objects and relations not represented in the original notation. Parameters to a function are displayed within the function’s box to permit their manipulation. A variable reference is displayed in its containing procedure as a separate object, thus allowing the programmer to manipulate either the variable reference or the variable declaration. Modules are also included (in the sense of Common Lisp’s packages [15]). Since Scheme’s scoping rules allow functions and variables within other functions and modules, such objects are drawn in a nested manner. To assist the tool user in identifying where a variable is referenced, a variable reference is graphically connected with the variable declaration it references.

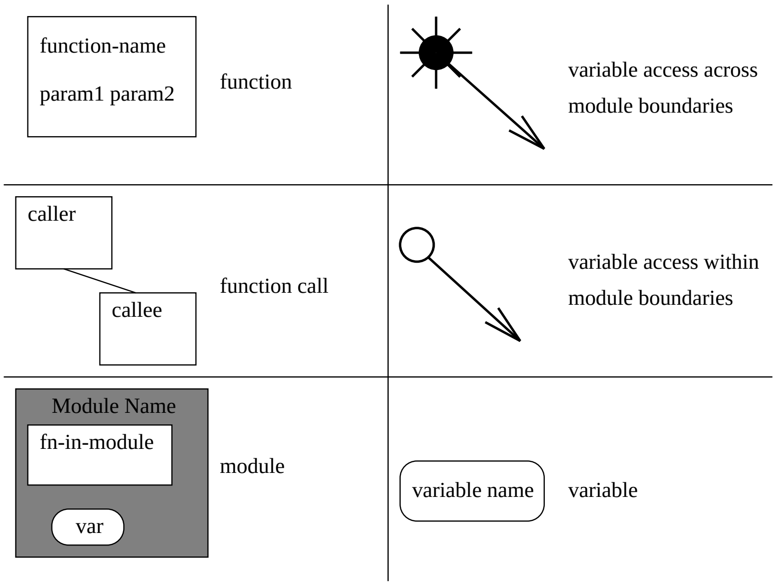

Fig. 2. Our notation for describing structure, derived from “Structured Design” [16]. The variable access symbols exist within functions, and represent variable accesses made by the function. Variable accesses are always graphically connected to the variable they represent.

In addition to specifying structural objects and relations from the source code, the notation used for restructuring can introduce relations and objects that highlight information pertinent to a particular design methodology. For example, in our notation, intra- and inter-module variable references have distinct symbols in order to highlight inter-module variable references, which in some circumstances are unacceptable in a methodology emphasizing data encapsulation.

4 Prototype Design and Implementation

The structure chart is automatically derived from the program text, guaranteeing that the two are always consistent. As the text is parsed into the chart, the tool builds mappings between the text objects and the objects displayed on the screen, which later support mapping transformations on the chart to transformations on the text.

With our prototype user interface, the software engineer can directly manipulate functions, variables, variable references, and modules. An object is renamed by editing its displayed name. Other transformations are applied by the engineer picking-up, dragging, and dropping an object using the workstation's mouse. The transformation performed is chosen in the tool through a table lookup keyed on the type of the object being dragged and the type of the object it is dropped onto. Any information that cannot be inferred from the action is prompted for in a dialog box. The selected transformation is then applied in the text-based transformation tool. When the transformation completes, the structure chart is updated to reflect the changes to the text. Since the tool uses the types of the object being manipulated and the target object to decide which transformation to perform, fewer additional parameters must be explicitly specified to perform a transformation as compared to the text interface (see Fig. 1).

The restructuring tool is written in Allegro Common Lisp, and uses the Picasso Application Framework [14] to implement the graphical user interface portions of the code. The tool runs on a Sun Sparcstation 2 workstation. Currently under development are module-level transformations and data-driven inference of unspecified parameters. Figure 3 shows a black-and-white translation of a color screen image of the structure manipulation interface for the tool.

5 Example

To demonstrate the nature of the design-level restructuring tool, we describe some of the operations that convert a functional decomposition of the KWIC (Key Word In Context) index production program into a data-encapsulated decomposition of the program [12]. (The transformations are described in greater detail in [4]). KWIC takes a text file as input and produces the list of words appearing in the document, with the context of each word (the rest of its enclosing line, circularly shifted to place the keyed word at the beginning of the line) appearing with it.

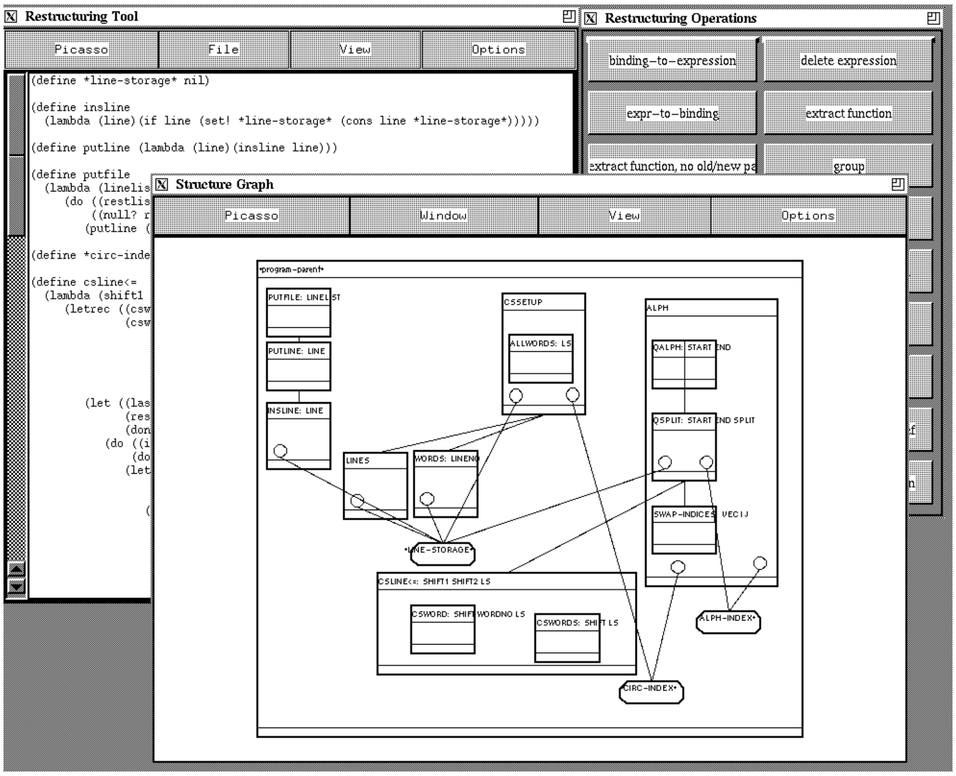

Fig. 3. Structure-oriented graphical user interface for Restructuring Tool. The upper-left window again holds the text representation of the program for reference and manipulation. The window in the center holds a structural view of the system.

Figure 4 presents the structure of KWIC implemented in a functional decomposition. The program represented by this structure is written in Scheme and is about 300 lines long. This version is divided into four modules, each performing a major subtask. The Input module reads lines of text into an internal array, the Circular Shifter module shifts the line's contents to show the keyword followed by the words in context, the Alphabetization module sorts the circular shifts, and the Output module produces the keyword list. Each module shares data with succeeding modules using shared arrays—`*line-storage*`, `*circ-index*`, and `*alph-index*`. Since we are encapsulating data representations, we must eliminate inter-module variable accesses, which are highlighted in our customized version of the notation. Since each module uses the results of previous modules, there are many inter-module variable accesses, and the tool user is visually alerted to these accesses, quickly ascertaining which modules are tightly coupled with respect to representation choice.

It is first noted that `*line-storage*` is directly accessed by many modules. The first task, then, is to hide the `*line-storage*` array in its own module. A

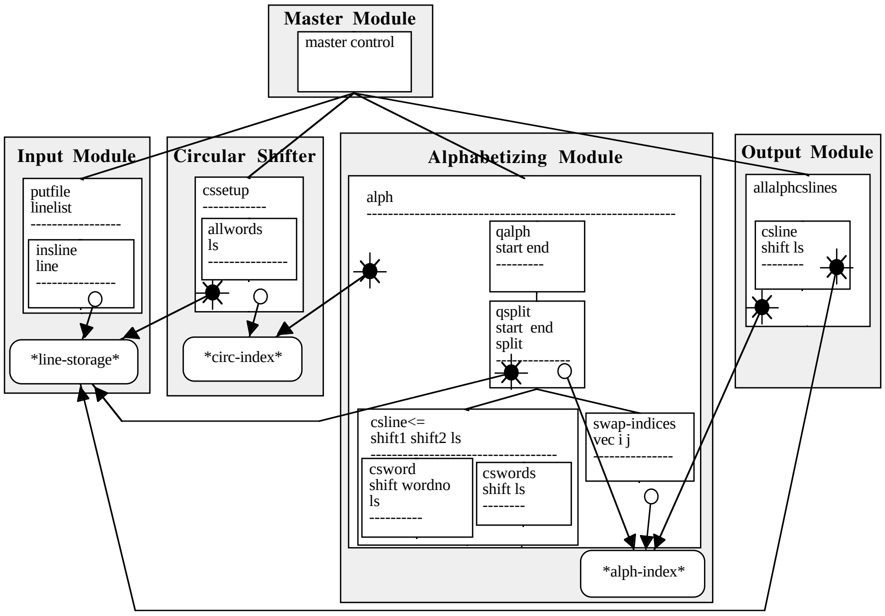

Fig. 4. Functional decomposition of the KWIC indexing program.

new module is created for the line storage by choosing a command from a menu. We then change the module of `*line-storage*` to the Line Storage module by selecting the `*line-storage*` variable declaration object and dragging it into the new module. When the transformation is mapped to the text-based restructuring tool, it checks that the modification will not introduce name conflicts or affect dependencies on the variable. The `insline` function (within `putfile`) is strongly related to `*line-storage*` so it is moved out of `putfile` and into the Line Storage module in the same fashion. Figure 5 shows the structure of the program after these modifications.

The references to `*line-storage*` within the function `cssetup` are inside common pieces of code that return the number of lines in the array, and the number of words in a given line. These should actually be top level functions. We create the new functions, `lines` and `words`, by dragging a variable reference out of `cssetup` and into its destination, the Line Storage module. The tool then pops up a window containing a text view of `cssetup`, and prompts the user to select the code to be the body and parameters of the new function. The transformation is then performed. Figure 6 shows the new structure.

Further restructuring creates new functions from inlined expressions, inlines array arguments to function calls, and changes the scope of objects. The resulting structure appears in Fig. 7. Not only has the notation aided perceiving the prob-

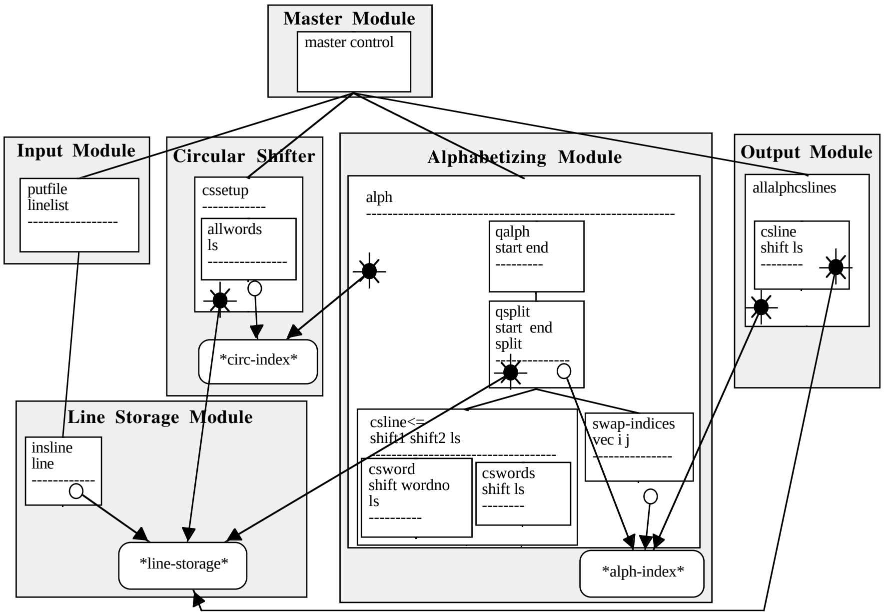

Fig. 5. First step in converting the functional decomposition of the KWIC example from a functional decomposition (Fig. 4) to a data-encapsulated decomposition. (Fig. 7) At this point, the Line Storage module has been created, and the insline function and `*line-storage*` variable have been moved into it.

lems in KWIC, but the direct manipulations on the notation also restructured the program text in lock step, in the process assuring that the restructuring is feasible and the meaning of the program is unchanged.

6 Limitations of the Current Approach

We find our notation and interface simplify the task of restructuring. However, the current approach and its implementation have a number of limitations, which are topics for current and future research. Three of these limitations are related to manipulating both the design notation and the program text. First, restructuring frequently involves abstracting new design elements from fragments of the program's implementation, such as functions from inlined expressions. To maintain the correspondence between the design and implementation, introducing a new design element such as a function requires linking it to the appropriate code fragment in the implementation. Creating a new function requires either the programmer to specify the code fragment to extract (using a view of the program text), or the restructuring tool to infer the code fragment based upon

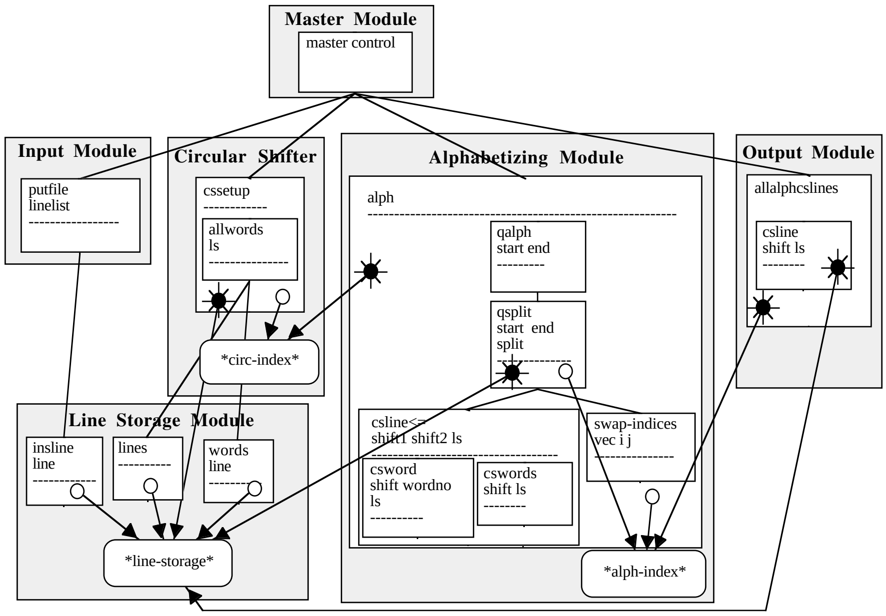

Fig. 6. The accesses to the variable `*line-storage*` from within cssetup are turned into functions by extracting the expression containing `*line-storage*` and using that as the body of the new function. The functions can then be moved into the Line Storage module. The last reference to `*line-storage*` will be extracted at a later step.

objects in the design being manipulated and additional information from the user. If the tool infers the code to extract, the restructuring task can be done completely within the design view, but gives the programmer less control. A solution allowing either approach would be appropriate.

Second, by changing the focus from implementation to design—and text to graphics—some information that might be useful to the software engineer is lost. For example, two functions may be placed close together in the text because they informally share information, such as the algorithm they use. If the functions are placed far apart in the structure chart to produce a cleaner layout, the software engineer may overlook the subtle relationship between the functions. We need more experience to see if this loss causes difficulty for users or if we can eliminate the problem. Other restructuring operations cannot be performed within the structure chart because they are implementation-oriented. For example, converting a `let` scoping construct to a `let*` cannot be handled at the design-level. However, the textual interface properly provides the support for such operations.

The third problem is that currently the design view is derived directly from the text, rather than an independently created design document. So although the

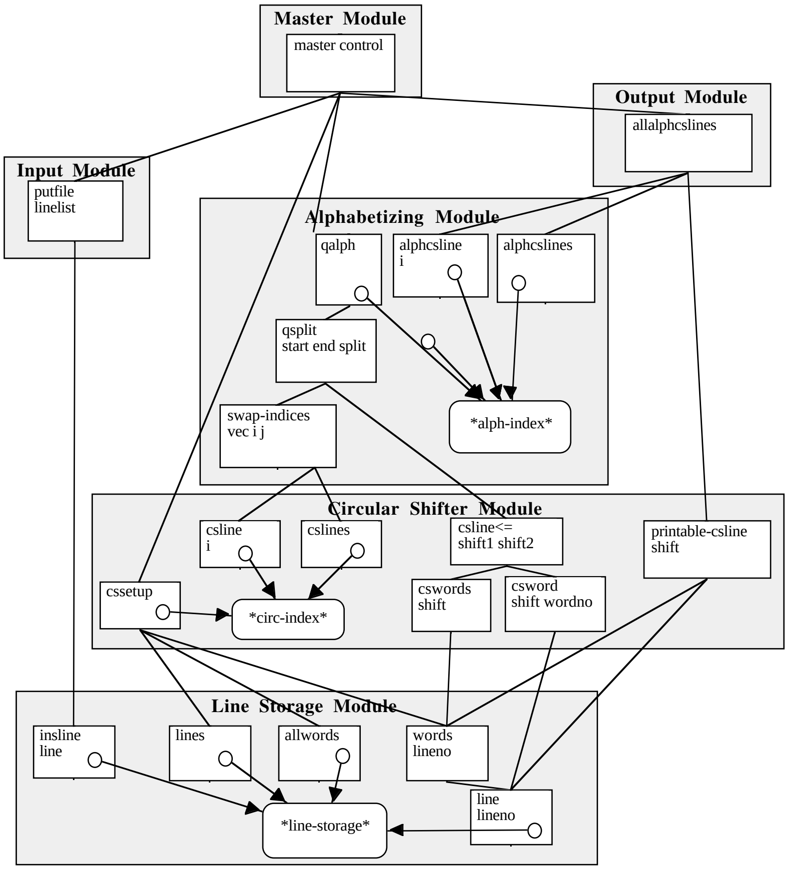

Fig. 7. Data-encapsulated decomposition of the KWIC program. This structure was generated by applying meaning-preserving transformations to the original structure shown in Fig. 4.

tool provides the right level of access, it does not support restructuring through a system's original design document. However, this should be possible for some real design notations, given that the appropriate links can be made between the design and the implementation to allow simultaneous restructuring of the two. The PegaSys system [10] (see Related Work) uses a simple technique for the user to establish links when they cannot be automatically derived.

Finally, since the information in a program is often not projectable into two dimensions, a graphical representation of a program can degrade into a spider web of connecting arcs for larger programs [1], a problem often encountered in program visualization. This problem can be mitigated by techniques such as those used in the Rigi program understanding system, which allows its user to remove extraneous information with elision and tool-assisted grouping of structural [11]. Similarly, automatically generating readable layouts of structure charts is difficult for large programs. Currently the tool uses a simple algorithm for initial layout, then allows the software engineer to interactively improve the layout. The engineer's adjustments then can be used in automatic layout decisions and saved for future restructuring sessions. More sophisticated layout techniques, such as force-directed heuristics [3] can reduce the need for user customization.

7 Related Work

There are many systems that support working graphically with 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 [11]. 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.

Our techniques benefit from a similar visual model, although ours is currently more primitive. 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, not allowing 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 [13] 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 [7]. 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 [10]. 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 connect the visual formalisms to the implementation. Once the visual formalisms are entered and connected to the implementation, PegaSys can proceed with the verification process.

The formal visual descriptions in PegaSys are basically design elements, and substantially assist program understanding, independent of the additional automation. As with our system, the connection to the implementation provides substantial manipulative power. PegaSys's 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.

A complete system for design-level manipulation of programs should support developing descriptive views, formal analysis, and direct manipulation of the design and implementation in concert. The similar designs of Garden, Rigi, PegaSys, and our system suggest that this integration is possible.

8 Conclusion

Tool-assisted program restructuring can improve the structure of a program to lower the excessively high cost of maintenance. However, a text-based restructuring technique forces the tool user to work with an inappropriate representation of the program. By displaying and restructuring the program at the design level, the software engineer can work directly with the concepts to be manipulated, not their implementation.

Our graphically based restructuring tool promotes understanding and manipulating of these concepts by displaying the structure of the program, directly representing the relations between components, and supporting restructuring through direct manipulation. Customization of the notation allows highlighting structural elements pertinent to a particular design methodology. By linking the design notation with the program text, feedback on the full consequences of a restructuring is immediate. Our prototype demonstrates that high-level, program structural redesign is feasible.

Acknowledgements

We thank David Notkin for discussions on this subject and comments on an earlier draft of this paper. We also thank the Picasso group at the University of California, Berkeley for their help in using the Picasso environment.

References

1. Fredrick P. Brooks Jr. No Silver Bullet: essence and accidents of software engineering. *IEEE Computer*, 20(4):10–19, 1987.
2. R. K. Dybvig. *The Scheme Programming Language*. Prentice-Hall, Englewood Cliffs, NJ, 1987.
3. Peter Eades. A heuristic for graph drawing. *Congressum Numerantium*, 42:149–160, 1984.
4. William G. Griswold. *Program Restructuring to Aid Software Maintenance*. PhD thesis, University of Washington, 1991.
5. William G. Griswold and David Notkin. Computer-aided vs. manual program restructuring. *ACM SIGSOFT Software Engineering Notes*, 17(1), 1992.
6. William G. Griswold and David Notkin. Automated assistance for program restructuring. *Transactions on Software Engineering and Methodology*, 2(3), 1993.
7. David Harel, Hagi Lachover, Amnon Naamad, Amir Pnueli, Michal Politi, Rivi Sherman, Aharon Shtull-Trauring, and Mark Trakhtenbrot. STATEMATE: a working environment for the development of complex reactive systems. *IEEE Transactions on Software Engineering*, 16(4):403–414, 1990.
8. M. M. Lehman and L. A. Belady. *Program Evolution: processes of software change*. Academic Press, London, 1985.
9. 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.
10. Mark Moriconi and Dwight F. Hare. Visualizing program designs through PegaSys. *IEEE Computer*, 18(8):72–85, 1985.
11. H. A. Müller, S. R. Tilley, M. A. Orgun, B. D. Corrie, and N. H. Madhavaji. A reverse engineering environment based on spacial and visual software interconnection models. In *Fifth ACM SIGSOFT Symposium on Software Development Environments*, pages 88–98, 1992.
12. D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972.
13. Steven P. Reiss. Working in the Garden environment for conceptual programming. *IEEE Software*, 4(6):16–27, 1987.
14. Lawrence A. Rowe, Joseph A. Konstan, Brian C. Smith, Steve Seitz, and Chung Liu. The PICASSO application framework. In *14th ACM Symposium on User Interface Software and Technology*, pages 95–106, 1991.
15. Guy L. Steele Jr. *Common Lisp the Language*. Digital Press, 1990.
16. W. P. Stevens, G. J. Myers, and L. L. Constantine. Structured design. *IBM Systems Journal*, 13(2):115–139, 1974.
17. Mark Weiser. Source code. *IEEE Computer*, 20(11):66–73, 1987.

This article was processed using the LaTeX macro package with LLNCS style