Tool Support for Planning the Restructuring of Data Abstractions in Large Systems

William G. Griswold, Member, IEEE, Morison I. Chen, Robert W. Bowdidge, Jenny L. Cabaniss, Van B. Nguyen, and J. David Morgenthaler

Abstract—Restructuring software to improve its design can lower software maintenance costs. One problem encountered during restructuring is formulating the new design. A meaning-preserving program restructuring tool with a star diagram manipulable visualization can help a programmer redesign a program based on abstract data types. However, the transformational support required for meaning-preserving restructuring is costly to provide. Also, programmers encounter comprehension and recall difficulties in complex restructuring tasks. Consequently, transformations were replaced with visual and organizational aids that help a programmer to plan and carry out a complex restructuring. For example, a star diagram manipulation called trimming was added, which mimics the way that basic restructuring transformations affect the star diagram display, allowing a programmer to plan a restructuring without depending upon restructuring transformations. With the ability to annotate trimmed star diagram components, plans can be recorded and later recalled. Programmer-controlled elision was added to help remove clutter from star diagram views. We implemented a star diagram planning tool for C programs, measured its elision capabilities, and performed a programmer study. We found that elision is effective in controlling star diagram size, and the study revealed that each programming team successfully planned its restructuring in rather different, unanticipated ways. These experiments resulted in important improvements in the tool’s software design and user interface.

Index Terms—Program restructuring, re-engineering, software maintenance and enhancement, abstract data types, modularity and information hiding, graphical user interface design, programmer study.

1 INTRODUCTION

Leintz and Swanson found that software maintenance can account for 70 percent of a software system’s lifetime cost [27]. Boehm documented an Air Force project in which the development cost was $30 per line, but the maintenance cost was as high as $4,000 per line [6]. Much of these costs can be attributed to poor software structure [4].

One way to lower software maintenance costs is to restructure the system into a more modular form while preserving the original functionality [22], [35], [25]. By isolating the code pertaining to a design decision within a module, changes to that design decision can be applied locally, and hence at lower cost [37]. An important special case of modular design is hiding a data structure and the computations that maintain its invariants behind a module interface [37], [28].

Performing a restructuring to encapsulate a data structure can be difficult when working with only the source code. Most of the source is unrelated to the data structure and the computations on the data structure are usually widely dispersed in the software, thus complicating the identification of all information required to design an appropriate interface. Often, programmers use a text searching tool such as grep to locate direct references to a data structure or global variable. However, grep’s line-based lexical orientation often yields false matches, includes irrelevant information, misses surrounding context, and cannot easily match all variables of a given type. Even with such support, a programmer must keep careful records in order to correctly perform the data encapsulation task. To address these problems we invented the star diagram, a hierarchical, graphical visualization of the computations on a data structure. It was originally applied as the user interface to a meaning-preserving restructuring tool [7], [10], [9]. Although this approach helped programmers to restructure small programs [8], there are a number of problems in scaling up to the restructuring of large programs. First, it is nontrivial to scale up the whole-program data-flow analysis that the tool uses to ensure that restructuring transformations are meaning-preserving [2]. This problem is exacerbated by the fact that a typical large system is implemented in several programming languages with widely differing syntaxes and semantics. Second, star diagrams built on large, complex programs can be quite large and difficult for a programmer to effectively assimilate [24]. Third, a restructuring task may be sufficiently complex that it would be unwise to begin restructuring before an overall plan has been worked out. However, the restructuring tool,
even with the star diagram interface, provides insufficient support for helping a programmer organize such a plan without performing such a restructuring [8].

We claim these scale issues can be mitigated by omitting meaning-preserving transformations and instead providing star diagram manipulation operations that a programmer can use to record restructuring plans and hide unneeded details. Additional navigational support can help a programmer perform a restructuring based on a recorded plan. Although the programmer is left with the task of restructuring with an editor or related means, the support provided can be made acceptably fast and robust, and star diagrams themselves can be used to reason about the correctness of certain aspects of the restructuring. Omitting transformational support also reduces the difficulty of implementing integrated support for multilanguage restructuring.

To demonstrate the above claims, we implemented a star diagram planning tool for C programs called CStar, performed a programmer study on a 28,000 line program, and separately measured its ability to hide unneeded details on programs up to 78,000 lines. The programmer study revealed that programmers were largely successful in planning their restructurings, that the tool was amenable to a number of styles of usage, and in fact programmers discovered new techniques for managing scale and for reasoning about the correctness of their changes. The scalability measurements revealed that programmers must aggressively elide unwanted detail from star diagrams to achieve sufficient size reductions. As a consequence, details hidden early in the planning process may need to be selectively re-exposed later.

Replacing the support for restructuring transformations with planning support results in greater interaction between the tool and the programmer over a larger body of program data. In essence, the complexity of transformations has been replaced by more complex user interaction. A principal insight from this research is that moving the complexity changes the requirements for program performance and user interaction, entailing changes in the tool’s software design and user interface. As a result, we were led to replace the tool’s pipeline design with a layered design in order to provide efficient hiding and unhiding of details in star diagrams. Using HCI methods, we learned that the user interface for CStar needs to maximize the visibility of program information while minimizing the number of windows and clarifying the relationships among those windows. The user interface was subsequently redesigned with a tiled, button-oriented organization and simple heuristics to reclaim windows that are no longer needed.

These techniques and insights may be generalizable to other program maintenance tools. For example, features provided in Rigi [32] can be used for elision of irrelevant detail and emulating code transformations on its visualization. The difficulty of coping with multilanguage programs is probably not unique to automated transformations for restructuring, motivating alternatives to heavy automation for assisting other maintenance activities. Our results also suggest that scaling up tool support for program maintenance and evolution requires attention to all aspects of tool design, not just isolated improvements.

The following section introduces the star diagram concept, its initial application to meaning-preserving program restructuring, and the problems encountered with this approach. Section 3 describes our proposed solutions, which are discussed by means of an example in Section 4. In Sections 5 and 6, focus is on evaluation, describing our programmer study and scalability measurements. Sections 7 and 8 discuss the software design and user interface improvements motivated by these experiments. In Section 9 we discuss how other tools facilitate planning. We close the paper (Section 10) with general remarks about scaling up programming tools and discuss future work.

2 THE STAR DIAGRAM

2.1 Application to Meaning-Preserving Restructuring

The star diagram was originally developed as the user interface for a meaning-preserving restructuring tool for imperative Scheme programs [7], [10], [9]. Fig. 1 shows the main star diagram window with a text window partially hidden behind it. The large white area contains the star diagram under consideration. The white buttons directly above are the transformations that can be applied through the star diagram interface; additional transformations are available through the text interface. The column of white boxes to the left of the star diagram is the area that displays the functions that the programmer declares to be the public interface to the newly created module.

Focusing on the star diagram itself, the diagram’s root at the far left represents all the instances of the variable to be encapsulated, in this case variable *line-storage*. The children of the root represent the operations and declarations in the program that directly reference the variable. The children of the next level represent operations that reference the previous level’s computations, and so on. A node having a stacked appearance indicates that two or more pieces of code correspond to (perhaps) the same computation. For example, the bottom node in the first column of children, labeled list-ref, represents all computations of the form (list-ref *line-storage* expr). The children in the last level, shown as parallelograms, represent the functions that contain these computations.

The resulting tree contains all of the computations that refer directly or indirectly to the data structure, and omits computations not related to the data structure. Because these computations are clustered according to similarity, a star diagram both collapses redundant computations into a single stacked node and provides a hierarchical presentation of each kind of computation on the tree. This presentation can help a programmer identify the computations that should serve as the implementations of new abstract data type operations. For example, a node near the left of the tree is an expression or statement that might implement a “low level” operation, encapsulating little more than the data structure representation itself. A node farther to the right might implement a “high level” operation. Moreover, a stacked node appearing farther to the right in the tree may draw the programmer’s attention due to its suggestion of nontrivial redundancy in the source code.
Each node in a star diagram is linked to a text view of the source so that underlying details are readily accessible. Double-clicking on a node displays and highlights its corresponding source code in the program text view. In the case of a stacked node, the programmer can navigate to each similar occurrence through a menu selection.

To apply a restructuring transformation to the program, the programmer clicks on a node, then clicks on the transformation’s button to apply it to the selected code. The tool first performs queries on the program that determine all the computations that may be semantically affected by the chosen transformation. It then performs semantic checks to ensure that the chosen transformation will leave the program’s functionality unchanged (e.g., scoping is not violated and the order of dependent assignments are not changed). If the check is successful, the tool performs the transformation on the source code representation and updates the star diagram to reflect the changes.

Transformations appropriate for performing data encapsulation in the star diagram include Extract Function, Inline Function, Extract Parameter, Inline Parameter, Rename Function, Move into Interface, and Remove from Interface. Transformation Extract Function is a central transformation. It creates a new function from existing code and replaces the existing code with calls to that function.

Suppose that after perusing the diagram shown in Fig. 1, it is desired to encapsulate all computations of the form (length (list-ref *line-storage* expr)) in a new function words-on-line. These computations correspond to the length node appearing in the second column of children. The transformation is invoked from the star diagram by clicking on the node stack—its border is darkened in Fig. 1 and one of the matching instances is shown in the text window above—and then clicking the Extract Function button. A dialog appears with a summary of each computation associated with the node stack, and the programmer selects which ones are to be replaced by a call to the new function. The tool then presents a dialog for specifying the function’s name and the parts of the computation that will become its arguments. Finally, the tool checks that the chosen computations are in fact behaviorally identical and that the transformation’s changes will preserve the functionality of the program. If the check fails, the reason for failure is reported to provide guidance on an alternative approach. In this case the check succeeds, so the tool performs the transformation and updates the star diagram to reflect the changes to the program (Fig. 2). The programmer can then click on the new function definition node in the diagram and click the Move into Interface button, removing the nodes corresponding to the definition and its body from the star diagram and placing a reference to the definition in the interface panel to the left (Fig. 3). This final action helps the programmer keep track the task’s progress by showing both what has been completed in the function list and what remains to be done in the star diagram view.

Repeating this basic sequence—interleaved with other transformations that prepare the code for function extraction—results in an empty star diagram, signaling that every computation on the data structure has been abstracted as a module operation.

2.2 Evaluation of the Approach

2.2.1 Difficulties in Providing Meaning-Preserving Restructuring

Implementing scalable data-flow analysis and alias analysis for the semantic analysis component of meaning-preserving
transformations is complicated, although possible [2], [31]. Another problem is that a large software system is likely implemented in multiple languages, requiring the implementation of multilanguage transformations and perhaps multilanguage data-flow analysis. A large project can have specialized languages for user interface description, database access, and domain-specific functions. Performance-critical parts of the system may be written in assembler. Also, popular languages like C and C++ support a macro language that provides conditional compilation and in-line macros that are not syntactically compatible with the programming language, and are intrinsically difficult to understand or manipulate semantically using traditional methods. As a noncommercial example, the tool described in this paper, Cstar, is implemented in six languages: C++, C, flex, bison, Tcl/Tk, and the C preprocessor. Popular macro-free languages like Java give hope that the macro problem may diminish over time, but program generators (e.g., bison, JavaCC) are becoming ever more popular. Moreover, many projects use in-house languages, complicating the production of off-the-shelf restructuring tools. Although none of these problems is overwhelming, and not all systems manifest them, together they suggest that industrial-strength meaning-preserving restructuring support may be impractical.

2.2.2 Experience with Star Diagrams and Restructuring

Several insights arose out of our experience with the restructuring tool, which includes two studies. To evaluate the concept of a meaning-preserving restructuring tool driven by a star diagram graphical interface, we performed a quantitative scalability study of star diagrams [9] and a qualitative study of pairs of programmers using one of a variety of tools [8].
The scalability study examined several star diagrams for large C and MUMPS programs, revealing that a star diagram for a widely used data structure can be unacceptably large. For example, a star diagram for all variables of type `struct buffer` in GNU Emacs 19.22 [39] contained 3,741 nodes and was 246 inches high when printed on paper. Such a star diagram is time-consuming for a programmer to assimilate for the purposes of planning a restructuring. Interestingly, the information generated by a `grep` on a variable (`grep` is awkward to use for all variables of a type) is roughly the same size as the star diagram for the same variable. It seems that the compression gained from clustering in star diagrams is offset by presenting a hierarchy of clusterings. However, star diagram users have the option of examining only the first level of clustering, which is invariably more succinct than the `grep` information.

The programmer study used systematic observational techniques [41] modeled after Flor’s laboratory studies of organization within programmer teams [18]. Flor’s approach of observing a pair of programmers working together, known as constructive interaction [29], was used in the study because it provides a more natural way to induce programmers to verbalize their thoughts than single-person think-aloud methods. Working dialogue also reveals how the programmers perform problem solving.

Six teams were observed for the experiment. Two teams used standard Unix tools such as `vi` and `grep`, two teams used the text-based meaning-preserving restructuring tool, and two teams used the same restructuring tool augmented with the star diagram interface. A small amount of training was given to the restructuring tool teams to familiarize them with basic tool features. Each team was given a 150 line Scheme program in a single file and asked to perform a data structure enhancement on it. However, each team was instructed to first restructure the program as necessary to ensure that the change would be modular. Using such a small program prohibited observing many scale issues, but it provided the possibility for detailed comparative observations across a number of toolsets. Each team’s programming session was limited to 2-1/2 hours to reduce stress on the programmers and to bound the amount of subsequent analysis.

Several insights arose from transcribing the videos, analyzing the transcripts, and distilling patterns of behavior. First, all teams employed techniques that balanced the tradeoff between performing complete changes and consistent changes; both are required for a correct change. Each toolset presents different tradeoffs to the programmers. The meaning-preserving restructuring toolsets ensure the completeness and consistency of restructuring transformations, which the Unix tools do not. However, the restructuring tools do not ensure a complete and consistent software design. On the other hand, the star diagram interface does provide a high-level mechanism for formulating such a plan by providing a visual hierarchy of computations that a programmer can inspect in a systematic fashion. The teams not using the star diagram interface adopted more complicated strategies for ensuring completeness and consistency.

In this regard, a strategy adopted by one star diagram team is worth noting. The team wanted to record design ideas prior to restructuring, even for this small program. In particular, more than once they created a function and moved it into the module’s interface in order to record a tentative design idea. The team was aware that the transformation sequence might have to be undone later with an inverse transformation, but it had the effect of changing the star diagram view (i.e., removing a path and giving it a name) to help recall the design idea.

Using restructuring transformations for design recall is time-consuming because the programmer must specify the function’s interface and the tool must perform data-flow analysis to verify that the transformation is meaning-preserving. Moreover, not every design idea is best represented as a function. These issues are magnified in larger programs due to the larger number of abstractions and the greater cost of data-flow analysis.

As a final, independent observation, a star diagram in the Scheme restructuring tool allows viewing computations on only a single variable. However, an abstract data type might consist of many components. For instance, a linked list might be represented by two variables: a pointer to its first element and a pointer to its last element. To properly encapsulate this data structure requires that both pointers be viewed in the star diagram [9]. Also, if there are many linked lists in a program, all need to be viewed in order to abstract them all into a single module.

### 3 Adapting the Star Diagram Concept to Large-Scale Restructuring

The unscalability of meaning-preserving transformations and the emergence of planning behavior by programmers performing even small restructurings led to the idea of providing direct support for planning restructurings in lieu of support for transformations. Programmers performing a large restructuring activity may make dozens of interrelated code changes to produce a coherent redesign, and they can benefit from assistance in the formulation and recording of a comprehensive plan to assist recall and avoid oversights. By augmenting this planning support with additional navigational mechanisms, a restructuring plan can guide a programmer to all places in the code that require change.

Such support could prove useful in planning complex restructurings regardless of whether the programmer performs the restructuring with an editor or with a restructuring tool. However, a planning mechanism also could be used for activities other than restructuring. For example, a phased inspection [26] of data structure invariants could be facilitated by a star diagram view. If problems were detected in such an inspection, changes would be necessary, but they probably would not involve restructuring.

Observing how programmers recorded their plan with function extraction suggested emulating the effects of restructuring transformations on the star diagram as a means of recording plans for a data encapsulation. The details of this emulation approach, the solution to the problems of viewing large star diagrams, and the extension of star diagrams to incorporate multiple variables are discussed below. These mechanisms are demonstrated on a running example in Section 4.
3.1 Emulation of Basic Restructuring Transformations

To allow the programmer to record a design decision without restructuring, we introduced support for mimicking the extract-function-then-move-into-interface restructuring sequence that programmers used in the study. Recall that in the Scheme tool, Move into Interface removes the path from a selected function node back to the root and places a textual annotation of it in the interface panel. In CStar, this trimming operation is extended to be performable on any node, with the implication that the code associated with the removed node corresponds to a key part of the data abstraction being planned, such as the implementation of a public operation. Since the trimmed node (and associated path) has not actually been restructured into an operation, it lacks a name that is suggestive of its role in the abstract data type. Consequently, CStar allows the programmer to annotate the trimmed path with a textual entry, which might describe the associated abstraction or how the associated code is to be restructured. A trimmed path can be viewed in isolation as its own star diagram. It is easy, then, for a programmer to later review a trim with its annotation and then navigate to the associated text to make the planned changes.

3.2 Elision of Uninteresting Nodes

In our programmer study on the star diagram restructuring tool, programmers tended to view a star diagram from left to right, starting at the root [8]. However, they sometimes used the function definition nodes at the far right to identify the context of usage. Aside from the function nodes, then, it appears that the right side of a diagram is not required much of the time and can be hidden using depth elision until needed. Variations of this approach are found in graphical class browsers such as Sniff [5].

Depth elision not only removes a significant number of nodes in a diagram, but it also permits a more compact layout of the remaining nodes when using particular off-the-shelf tree drawing algorithms. For example, the standard Tcl/Tk tree widget [11] uses an algorithm that places the children of a node as close together as possible [30], meaning that two siblings with many children will be placed far apart from each other. Eliding the children allows the siblings to be placed close together, making for a more compact layout that permits easier comparison of the siblings.

Path trimming also serves as a mechanism for permanently or temporarily hiding uninteresting subtrees of a star diagram. However, trimming and depth elision are not adequate for many situations. Sometimes there are too many function nodes for a compact view, or the uninteresting nodes may be more properly identified by a node property other than its depth in the tree [23]. For example, a star diagram can be crowded by nodes denoting heavily nested if-then-else statements. Consequently, we developed a mechanism that allows hiding nodes based on a variety of properties, including their syntactic class, depth in the tree, stacking characteristic, or a string match on their labels.

3.3 Extension of the Star Diagram’s Root Concept

To permit visualizing data structures that incorporate many data elements, we introduced support for the specification of multiple variables and expressions as the root of the star diagram. A composite root can be specified using a combination of methods: multiple variables can be selected from the text view of the source, any number of existing star diagrams can have their roots unioned, or a variable’s type can be used to specify a root that contains all variables of that type. Although not supported in CStar, any program query that produces a set of related identifiers, expressions, statements, or declarations could serve as the basis for a star diagram root set.

3.4 Enhancements to Navigation and the Star Diagram Itself

A key task of a programmer who is investigating a star diagram is deciding which nodes constitute abstract operations of the data type and how their associated code should be restructured to create those operations. For a stacked node, this requires comparing all the similar code fragments from throughout the program to decide which parts of the code fragments are common and which need to be abstracted away as arguments. To better facilitate comparing all the code associated with a stacked node, we introduced a grep-like view of stacked nodes that is called up by double-clicking on a node. Clicking on any entry in this grep view opens a text view shows the associated fragment highlighted in context.

Finally, because larger programs consist of many files—and a file’s name often suggests the role of its contents in the overall system—we added file nodes to the right side of the diagram.

4 A Planning Scenario

To provide a more detailed, dynamic perspective of the ideas introduced in the previous section, we present a part of a session in which CStar is used to plan the restructuring of a dungeon exploration game program called saadventure [1]. This version of saadventure is a 1,700 line C program (excluding comments and blank lines), although it was originally written in BASIC.

4.1 The Maintenance Task

A major enhancement is planned for the rooms data structure, a global array of room descriptions. In particular, the geometry of the rooms is going to be changed and the number of rooms may vary during the game due to the actions of the player. However, rooms is directly accessed throughout the program, so the enhancement is expected to be costly. Our goal, then, is to encapsulate the rooms array in a module that localizes the enhancements to that module, hence making them easier to perform.

We start CStar, which immediately displays a root window that provides all the tool’s project-wide operations. We select the appropriate directory and files for the project, and the files are loaded (Fig. 4). If we wished, we could save out this configuration with the Save button so that future work on this project could be resumed by using the Load option. The bottom of the project window contains an entry for each created star diagram, which includes a list of its roots and an optional annotation.
4.2 Early Exploration

To begin planning the encapsulation, we wish to build a star diagram based on the type of the variable `rooms` (Fig. 5). This is accomplished by first searching for the string “rooms” using the Search for a String in All Files option in the project window, which returns a grep-like result. Selecting a match that corresponds to a `rooms` reference navigates to the match’s location in the source code. From here we request that the reference’s type be included in the root set for a new star diagram, and then display the star diagram. Having chosen the type of `rooms` as the root means that all expressions of the same type as `rooms` are included in the star diagram. However, we notice that only references to `rooms` are included, indicating that `rooms` is the only variable that has its type. This suggests that there is just one `rooms` array and that the array is never referenced through another variable, including a function parameter (which would necessarily be of the same type unless `rooms` was first subjected to a type cast operation). Consequently, it is not necessary to create an interface for `rooms` that accepts it as a parameter; it can be completely hidden in the new module and referenced by the module’s procedures as a private module variable.

To begin our search for a small set of functions that intuitively convey the concept of manipulating rooms, we start looking for ideas in the first column of children from the root node. It is immediately obvious that the `rooms` variable is an array since 30 out of 36 accesses occur inside an `ArrayRef` node. Clicking on the `ArrayRef` node displays a grep-like list of lines containing the array references (Fig. 6). We can see that an array of `rooms` is being indexed by integers. There is a lot of information to assimilate, and nothing comes to mind in terms of an abstraction except an overly simplistic “get room” operation. We dismiss the view and begin to look for other ideas from the star diagram.

Because only a couple of the root’s children can be seen on the screen at once, we decide to apply depth elision to adjust the diagram’s width to three levels of nodes (Fig. 7). The diagram now fits on the screen, although too much of the diagram may be hidden for later parts of the task. However, at the top of the diagram we see a stack of five declaration nodes (PtrDec followed by Declaration). Curious, we select the node stack, and the grep-like view of the declarations appears (Fig. 8). One entry is the actual declaration of the `rooms` array, and the others are `extern` declarations that import the array into the other files of the program. It is interesting that the `rooms` array is actually a pointer to a `Room` structure, because it indicates that the `rooms` array is probably dynamically allocated. We are also concerned that the `rooms` pointer might be used in a more dynamic fashion than merely pointing at the beginning of an array.

Since we are encapsulating `rooms` behind a set of functions, we realize that the `extern` declarations will soon be obsolete. Consequently, we decide to remove this node (and the paths passing through it) from the diagram. This is performed by clicking the Trim Arm button appearing at the top of the star diagram window. Cstar moves the declaration node and associated paths from the diagram to the trimmed-node panel on the left. We then click the Edit Entry Annotation button above the panel in order to record our plans for the code associated with the node. We type “Declaration of rooms array and external declarations. Double-check that rooms is not changed after initialization. If OK, move declaration to new module file, and delete external declarations” (Fig. 9). As a consequence of this action, we have not only begun recording a concrete plan for encapsulating the `rooms` array, but we have also further shrunk the diagram so that the remaining components are easier to view.
Fig. 5. Initial star diagram for the rooms array in saadventure as it appears in CStar, the C star diagram planning tool.

Fig. 6. Grep-like view of the ArrayRef star diagram node.

Fig. 7. The elided rooms star diagram.
4.3 Identifying Part of the Initialization of rooms

Besides the 30 array references, there is only one other child of the root left in the diagram, labeled LHS of assn, which means that rooms appears on the left-hand side of an assignment somewhere in the source code. Since we are concerned about how rooms might change during program execution, and the array portion of the diagram is daunting, we decide to look into this assignment. The node itself indicates that there is only one assignment to rooms, which seems promising, but if it were in a loop or a frequently called function, then more than one array of rooms could be created. The function definition node appearing to its right—which is on-screen because of the elision—tells us that the assignment appears in the function ReadDataFile. It is probably an initialization routine. Clicking on the LHS of assn node, we are presented with the code itself (Fig. 10), which confirms our belief that this is initialization code. If we were not so sure, we could build a star diagram for ReadDataFile to find where it is called. Now that we are sure the assignment is an initialization, we know that rooms really “is” the rooms array, which will allow a more straightforward interpretation of the rest of the star diagram.

Since we have been investigating the initialization, we could try to figure out what module interface function we should have for initialization. In particular, do we want just the malloc call, do we want the whole ReadDataFile routine, or do we want something in between? However, right now we are more concerned with the geometry of rooms, and it seems that the array references must be directly involved. Before leaving the initialization, though, we trim it from the diagram to simplify the view and also record our (unfinished) plans for the assignment with the annotation, “Allocation; part of initialization. Find code that initializes the contents of array and decide if should be separate or combined.”

4.4 Abstractions for Room Operations

Now only the ArrayRef child node remains, and we can see that it is used in two record field accesses, as indicated by the Field:desc and Field:dir labels. We first examine the dir field accesses. We unelide the star diagram one level to get a look beyond this reference, but this shows little, and the Field:desc node is interfering with its layout. So instead we view this star path by trimming it into the trimmed star arm panel and viewing it separately (Fig. 11a). Since the child node of the Field:dir star node is an ArrayRef node, the dir field must be an array. A quick look at one of the print statements under the array reference (Fig. 11b) indicates that each integer entry represents a possible exit from the room. Among the 21 ArrayRef exit references, we see that:

1) their addresses are taken,
2) they are printed,
3) they are assigned to another variable (i.e., RHS of assn), and
4) they are compared for equality and inequality.

We notice that the consumer of the addressOf star node is an fscanf function call node in the function ReadDataFile. Examining the source code reveals that the fscanf call ini-
itializes the rooms array from a file. Thus, we can trim the `fscanf` node with the annotation, “To be grouped with the allocation operation for initialization” (Fig. 12).

Moving on, two comparison nodes about half way down the diagram stand out, a $\neq$ node and an $=$ node. The source code for the $\neq$ node is:

```c
if (rooms[currentRoom].dir[i] != 0)...
```

This code helps us determine that the expression is a test for a valid direction. We proceed to trim this node from the star diagram view and annotate it as the operation “IsValidDirection.” Interestingly, the equality comparison is also with 0, and we determine that this is a test for an invalid direction. We trim this node with the annotation, “IsInvalidDirection, consider combining with IsValidDirection.” It is notable that the star diagram’s global view facilitated finding the similarity between these two operations.

The next child we consider is a RHS of assn star node. The corresponding source code is:

```c
currentRoom = rooms[currentRoom].dir[noun - 1];
```
This computation sets the currentRoom variable to a new room after the player makes a move in a valid direction. Our first inclination is to move this star path to the trimmed-node panel as the “MoveTo” operation. However, this operation updates currentRoom, which is not part of our planned module. To explore whether currentRoom should be part of our module or part of its own module, we build a star diagram for currentRoom. Along the way, we find a similar variable startingRoom, and union it into the currentRoom diagram using the Combine Diagrams feature in the project view to create a star diagram with both of them as roots. Realizing that there in fact could be many such variables, we build a type-based star diagram for currentRoom. This diagram turns out to be huge because currentRoom is an integer, yet few integers appear to be room indices. Trying an alternative approach, we return to the project view to search for the string “rooms,” which indirectly identifies all integers that are used as subscripts to the rooms array. We then build a variable-based diagram for all the global variables that are identified, which is much more manageable yet still large (Fig. 13). Trying a few elisions, we find that excluding all statement-level nodes as well as function and file nodes provides a concise view (Fig. 14). After perusal of this diagram, we conclude that currentRoom represents the player’s current relationship to the rooms, and so is not intrinsic to the rooms abstraction. For instance, if this game were extended to have many players, the rooms data structure might not change, but there would be a current room for each player. Consequently, the “MoveTo” operation is part of the player abstraction. However, we still need to encapsulate this rooms reference, which—without the assignment—represents the “adjoining room” operation. We trim the ArrayRef node (not the assignment node), annotating it as “AdjoiningRoom(int room, int direction).”

4.5 Discussion

Both trimming and elision were used to create a succinct but still meaningful view of all uses of the rooms array. In a more complex example, more elision and node trimming would likely be required to produce a comfortably viewable star diagram, probably resulting in a view that is missing crucial information. However, the quick reversibility of these operations allows for flexible viewing strategies.

We were able to peruse the diagram to inspect related groups of code on a case-by-case basis. The grep-view displayed a summary of each member of a node stack, exposing similarities and differences, as in the case of the external declarations of rooms. The grep-view also supported navigation to a text view of the code, permitting the examination of nearby statements. By trimming a node and its associated paths and then giving it an annotation, we constructed a plan of the encapsulation. The trimmed node and its associated paths can be reexamined later as a star diagram, grep-view, or source code, assisting with recall, further analysis, or performing the planned change. Trimming “handled” paths from the diagram also focuses attention on the computations that have not yet been handled. The shrinking size of the diagram provides a visual cue of the task’s progress.

As new information was discovered in the process of the encapsulation, such as the special status of the currentRoom...
rentRoom variable, we could construct new star diagrams on the fly and merge them together as needed to plan a multivariable encapsulation. If we had decided that currentRoom or one of the other room indices should be part of the rooms abstraction, we would have built a star diagram combining the rooms and the index variables of interest. Such a big change to the rooms abstraction would have led us to reconsider the prior decisions. The low cost of identifying and recording design decisions in the planning tool—relative to actually restructuring—makes this backtracking palatable. On the other hand, the lightweight planning support has delayed the behavioral checking of the planned operations until the actual restructuring is performed.

5 User Study

Our previous programmer study spanning three toolsets identified the tradeoff between completeness and consistency as a way of understanding how programmers restructure their programs (See Section 2.2). The presence or lack of certain mechanisms lead programmers to a variety of strategies for planning and performing their restructuring changes. To better understand the impact of the added planning functionality and the lack of restructuring transformations on programmer behavior, we undertook an exploratory systematic observational study using an early version of CStar [24]. The study was planned at this intermediate stage to inform further design of the tool. The version used lacked support for specifying star diagram roots.
The term team is used here to indicate that the pair of programmers were working together towards a common goal, not that they work together regularly. However, each pair was well acquainted and seemed comfortable working together.

2. The term team is used here to indicate that the pair of programmers were working together towards a common goal, not that they work together regularly. However, each pair was well acquainted and seemed comfortable working together.

3. The example of this technique, described in Section 4 and shown in Fig. 11a, was introduced after the studies were completed.
much of the interface, team B created one of their key operations, \texttt{objectgetnth}, and modified the source code to call it. They then generated a new star diagram that not only had the \texttt{Objects} root, but also the new function so they could investigate all calls on it. Team B preferred not to use annotations, but since their planning and restructuring activities were interleaved, annotations were not so crucial to aiding recall. They did use the trimmed arm panel to keep track of the computations under consideration for each operation.

Team C’s approach was similar to A’s, but they tended to defer issues that initially confused them, resulting in multiple passes over the \texttt{Objects} star diagram. For instance, \texttt{Objects} is passed as an argument to functions \texttt{disarm} and \texttt{inititem}. As their plan developed, team C revisited these functions to ascertain, for example, if they might be part of the planned module or if the uses of the pointers in these functions violated the newest aspect of their plan. Toward the end, to help answer the latter question, they built a star diagram of the parameter in \texttt{disarm}, and quickly determined that it was in fact used only as a pointer to the entire \texttt{Objects} array, and not to an individual object in the array. A multipass strategy raises the question of how they recalled both intermediate and final plans. They used trimming with annotations for recording final decisions, relying on mental recall and a top-to-bottom visitation strategy in the star diagram to keep track of their progress. As soon as team C converged on their basic plan—but before making final decisions about the treatment of \texttt{disarm} and \texttt{inititem}—they began sketching out the new code in new source and header files.

5.2.4 Conceptualization

All teams treated nodes in the diagram as candidates for abstraction into a function. However, team B’s novel use of multiple roots to analyze the correctness of their changes suggests that they conceptualize the star diagram root as being “everything of interest,” and not just “all references to the data structure being encapsulated.” In hindsight, this team’s conception is not surprising, because once they had created a new function, the elements of interest naturally grew to include the new function and its uses.

5.3 Observed Problems

5.3.1 Lack of a Searching Capability

All teams had problems locating the declaration of \texttt{rooms}, since at the time CStar lacked a whole-program searching functionality other than the star diagram itself. Thus, the teams relied on Unix \texttt{grep} to find the declaration and to perform other searches over the \texttt{Omega} source code. These problems led to the addition of the current program-wide searching feature.

5.3.2 Lack of Automated Restructuring

Team B complained that it was tedious and error-prone to perform the actual restructuring by hand. They used Unix’s \texttt{sed} utility to apply their global changes, and it took a few tries to fashion a script that made the change correctly. Time ran out for the other teams before they could attempt the changes. Clearly some sort of transformational capability is desirable, although it is unclear how to do so without adding unacceptable complexity.

5.3.3 Manually Specifying Type-Based Star Diagrams

Confirming our expectation that programmers would want type-based star diagrams, the teams complained about the inability to quickly generate a star diagram for all variables of a particular type. Teams wanted this feature to observe how an object copied out of the \texttt{Objects} array is used. Although it was possible to manually specify all the variables for the root, the teams only explored this option briefly because it was (reasonably) perceived to be too much work to be worthwhile.

5.3.4 Selecting a Subset of a “Stacked” Node

Team A complained that CStar lacks a feature to select a subset of the members of a stacked star node for trimming. We observed the same problem in the \texttt{saadventure} example when trimming the external declarations, although the flexibility of annotations mitigate this problem. Interest-
ingly, the function extraction operation of the original Scheme star diagram tool supports this selectivity, but we overlooked it in our design of CStar. Because of the complexity of providing this feature in a planning tool and the ability of programmers to work around it, we decided to leave it out of later versions of the tool.

5.4 Observed Successes

5.4.1 Support for Planning

All teams successfully planned the creation of a set of functions to encapsulate the data structure in a limited period of time. (By successful, we mean that the plan, if carried out, would have succeeded in localizing the code involved in the proposed change.) No team’s planning was inhibited by the lack of restructuring capability, although the task does not capture the potential interaction of planning two modules prior to restructuring. Moreover, each team chose a different abstraction, suggesting that the star diagram is a mechanism for data encapsulation, not forcing a particular style of interface on a module. The wide difference between team A and the other teams can be attributed in part to team A’s experience.

5.4.2 Flexible Mechanisms for Coping with Scale

Not only did team A use depth elision and all the teams use trimming to scale their diagrams, but team B invented a new method on their own. Team B’s divide-and-conquer approach of moving a node into a subdiagram is effective because it provides a better layout and allows “stacking” several diagrams on top of each other by using the window system’s features. It also may be a more effective way of achieving elision than removing “uninteresting” nodes, since there may be more interesting subdiagrams than uninteresting ones.

5.4.3 Use in Correctness Analysis

The lack of automated restructuring in CStar also means that the correctness of restructuring must be independently checked, either by some form of inspection or software testing. Team B’s tactic of building a star diagram to investigate their restructuring changes suggests that programmers can use the tool’s features to assist in checking the correctness of their changes, somewhat mitigating the lack of fully automated checking.

6 Effects of Elision on Scalability

Since scale issues were not readily exposed by the techniques and subject matter used in the programmer study, we performed a separate quantitative study on the impact of elision and trimming on star diagram size. We focused on the relative effects of trimming and elision because a previous study identified major scalability issues and compared the size of star diagrams to the size of comparable grep queries (Section 2.2). For the measurements in this study, we used ubiquitous data structures in omega (described in the Section 5) and emacs, the terminal-based version of GNU Emacs version 19.34 [39], which contains 78,000 lines of C code.

The primary concern with star diagram size is the effort required to assimilate a large star diagram. Two influential factors are the volume of information in a star diagram and how much scrolling is required to view this information. Both are important to measure because a star diagram may paradoxically contain a large amount of information in a compact layout or a few nodes in an inefficient layout. To capture the volume of information, we count the number of nodes remaining in the star diagram. (Another measure, the number of paths, is also interesting, but harder to interpret.) To capture the complication of scrolling, we measured the heights of node declarations subtree and file nodes. Big diagrams like these probably provide more trimming opportunities, but identifying appropriate candidates could be more difficult, too.

As a measure of the potential benefits of trimming a path, we chose to trim the declarations in each star diagram at each of the elision depths. Trimming the declarations conservatively measures the effects of trimming, since presumably at least a few paths can be trimmed during encapsulation as a diagram reduction measure. Although the declaration subtree is often quite small, these variables are broadly exported, so their elision significantly simplifies the layout of file nodes. Big diagrams like these probably provide more trimming opportunities, but identifying appropriate candidates could be more difficult, too.

The results, shown in Figs. 16 and 17, suggest that both elision and trimming can provide substantial reductions. Trimming is directly effective at reducing height. Depth elision is most effective at reducing the number of nodes, particularly in larger diagrams, where it also can produce substantial height reductions. Function and file node elision
are extremely effective at reducing the height of a star diagram. The effectiveness of these elisions is of course dependent on the programming style for the application; if these systems were coded with large functions and just a few files, function and file node elision would not have been as effective—nor as necessary. However, because of the wide range of elision options—including string pattern matching on node labels—programmers are able to fashion an elision effective for their application.

Modest depth elisions on their own do not produce enough reduction to minimize scrolling and keep relevant data clustered together. Function and file nodes must be elided and aggressive depth elision of some kind is required. Such elisions necessarily hide valuable information, requiring the programmer to frequently change the elision options to include the information required at the time.

Despite the large reductions for *temacs*’ *current_buffer* data structure, the second-smallest diagram is awkward to view and the smallest does not provide much information. The problem is that the diagram’s greatest branching occurs at the first level of children, largely due to the fact that the data structure being visualized is a record containing in excess of 40 fields. We believe that such a star diagram can be more easily viewed by laying out the first-level nodes according to their class of access, such as placing all structure field accesses together, placing all function calls together, etc. [10]. Additionally, programmer-controlled node-stacking criteria could increase node stacking—such as stacking all printing operations together—to produce more compact diagrams [10], [16].

7 SOFTWARE DESIGN AND IMPLEMENTATION

Because CStar is intended for use on large programs, scalability of performance is a major concern. Although exhausting main memory is an issue for some tools and can require special handling [2], this has not yet been a concern for this tool. However, the time for computing and drawing star diagrams was a bottleneck in the version of the tool used in the programmer study (See Section 5). For example, eliding and redrawing a star diagram for *current_buffer* in Emacs required 20 min or more on a 160 MB Sparc 10/61, regardless of the amount of elision. Much of this time can be attributed to the interpretive overhead of Tcl/Tk [36], which is used for the computation, representation, and display of elided star diagrams. However, we also noticed that display times did not improve significantly when a star diagram was heavily elided, suggesting that algorithmic factors were playing a role. Although it may be acceptable to pay such costs occasionally, Section 6 suggests that frequent customization of elisions may be necessary in order to manage the tradeoff between the size of a displayed diagram and showing all the needed detail.

The version of CStar used in the programmer study [16], [24] was prototyped in stages, left to right, yielding a pipeline architecture (See Fig. 18). The first and second stages,
AST construction and star diagram construction are implemented in C++. Tcl/Tk provides the user interface components, including the third stage, which constructs and displays a star diagram using the Brighton tree widget [11]. With these first three components, there are two stages of star diagram construction—one in C++ called the internal star diagram and one in Tcl/Tk called the visualized star diagram—introducing some redundancy. This redundancy is difficult to avoid, since Tcl/Tk and its interface to C++ are not efficient enough to walk the relatively massive AST and construct a visualized star diagram from scratch. On the other hand, writing the user interface in Tcl/Tk is very attractive because it is high-level and flexible.

The fourth stage provides depth elision and trimming of the visualized star diagram. However, this stage is destructive to the visualized star diagram it consumes. In most instances this stage is relatively efficient, deleting nodes from the star diagram as specified by the selected elisions and trims.

Finally, navigational capabilities are provided, such as opening a grep window when a node is selected. Such operations do not fit cleanly in a pipeline style. Auxiliary mappings are constructed as each representation is built from the last, keeping track of the association between a visualized star diagram node and its internal star diagram node, and also from the internal star diagram node to its underlying AST nodes. To open a text window for a selection, then, the tool merely follows the pointers back from the selected object to the corresponding AST, unparses the AST for the containing compilation unit, and adds highlighting to the text that corresponds to the selected AST nodes.

Although this design grew naturally out of the incremental prototyping process, it has serious shortcomings in both performance and extensibility. First, we were dissatisfied at an intuitive level that the design was a mixture of translation-oriented pipeline stages and navigation-oriented mappings. Second, because two full star diagrams have to be built before any elision takes place, drawing a highly elided star diagram can take much more time than expected (although drawing time would be reduced to offset some of the loss). Moreover, because the fourth stage is destructive to the outputs of the third, when a programmer changes the elisions a new visualized star diagram is constructed in its entirety and then reelided. This is unacceptably slow if a programmer first selects extreme elisions to get a view of acceptable size and then gradually disables these to expose more detail as the planning process progresses.

One natural alternative to a pipelined architecture is a layered design. Layering itself, however, does not solve the problem of constructing two full star diagrams before elision can be undertaken. The tool could optimize some elisions by recognizing when a new elision will only remove more nodes than the current one (e.g., eliding to a greater depth), but this is often not the case. However, elision of the visualized star diagram need not be implemented by deleting nodes. Instead, the elided, visualized star diagram can be built directly from the internal star diagram by simply omitting nodes that are to be elided during the construction of the visualized star diagram. With this approach, reeliding does not require reconstructing a complete visualized star diagram, but only the parts the programmer wants. The programmer still has to wait for the initial complete construction of the internal star diagram, but a significant performance improvement is still realized by eliminating one full stage of star diagram construction. Moreover, because of the abstraction provided by the layered design, the internal diagram implementation could be changed to incrementally construct a star diagram without the knowledge of the upper layers. The revised architecture is shown in Fig. 19.

We expected that the new design might run more slowly in some instances, because of the more general elision features that were added in the process (i.e., property-based elision). However, we expected the layered design to run more quickly on larger star diagrams, since it makes one less pass over the visualized star diagram, not only reducing the number of instructions executed, but also improving cache and paging characteristics. On the other hand, we were concerned that the internal star diagram construction time might still dominate, since it required a full traversal of the AST, which is much larger than a typical star diagram.

To assess the performance differences between the two designs, we timed a variety of elisions on Omega’s objects variable (Fig. 20) and Emacs 19.28’s current buffer variable (Fig. 21) on the Sparc 10/61. On internal star diagrams

---

4. Although star diagrams are not displayed as trees, they are internally represented that way.
that contain a only few hundred nodes, the layered design does typically run more slowly than the pipelined one. However, on the diagrams that really matter—ones whose size runs into the thousands of nodes—the layered design is two to 10 times faster depending on the amount of elision. The layered design is faster on more aggressive elisions regardless of the size of the internal star diagram. Importantly, the layered design’s running time proportionally decreases as the number of nodes to be displayed decreases, so a programmer is not forced to wait unexpectedly for the drawing of a small diagram. In fact, to speed initial display times, the user interface was changed to first display a star diagram at maximal elision, reducing the initial display time to under 2 min for the current_buffer star diagram. Although the display times are substantially improved, they may still be too slow for some applications. Since virtually all of the computation and display of elided, visualized star diagrams is performed by interpreted Tcl/Tk code, replacing the Tcl/Tk implementation with a native windowing implementation or upgrading to the new Tcl/Tk 8.0 implementation [43] (which now performs compilation and supports types other than strings) would likely improve performance to acceptable levels.

8 USER INTERFACE DESIGN

Because of the increased interaction required to use CStar relative to the use of a meaning-preserving restructuring tool, the user interface can subtly but profoundly affect a programmer’s ability to plan and carry out restructurings. Although the programmer study in Section 5 yielded positive results in terms of how programmers exploited the basic functionality, a review of the video tapes revealed a
number of user interface problems that promoted frustration and other undesirable results [14]. Most notably, as programmers progressed in their work, the number of windows on the screen grew, leading to confusion when trying to find an old window. Also, inconsistent placement and access to functionality in different windows caused delays. For example, depending on the window, it might be closed through a pull-down menu on the left or a button on the right. Some functionality and useful data—like the elision menu and the trimmed arm panel—were hidden in pop-up panels that were sometimes forgotten or lost in the proliferation of windows (See Fig. 15). Some functions were misinterpreted because of where they were placed in the tool or the way the functionality was named. On top of all this, programmers frequently repeated simple actions such as selections in order to get real work done.

Being aware that these problems would only get worse as larger tasks were undertaken, we decided to redesign the original user interface using standard HCI principles. Four principles seemed most relevant to our application.

8.1 HCI Principles Relevant to CStar

8.1.1 The Visibility Principle

Relevant data and functionality ought to be as visible as possible to a tool user. This principle is predicated on the assumption that the human capacity for recognition is more powerful than recall [38, pp. 118–121]. The programmers in our studies experienced severe visibility problems due to the proliferation of windows and hidden functionality. Unfortunately, much of the visibility problem is inherent in the task: a star diagram planning tool is intended to help programmers to discover relationships amongst syntactically separated pieces of code. Because there is no restructuring engine, the programmer must also revisit all of these related program locations in order to carry out the actual restructuring changes. Other visibility problems were due to design. For example, on tool start-up only a tool bar was displayed. Each button on the bar invoked a pop-up window that was responsible for some aspect of the tool’s high-level operation.

8.1.2 The Consistency Principle

Things that are similar should look similar, and things that are different should look different [13, pp. 9, 21–24, 32, 101]. When successfully applied, a user interface is easier to understand because the user can reliably respond to visual cues of similarity and difference in the interface, aiding recall and also the ability to guess how unfamiliar parts of the tool might behave when used. The problem with dismissing windows is a prime example of inconsistency in CStar. The challenge in applying the consistency principle lies in making decisions about what “typical” tool users will judge to be similar, and what they will judge to be different.

8.1.3 The Clear and Concise Language Principle

Names or images given to functions and displayed data should be easy to understand without cluttering the interface [13, pp. 21–24]. CStar’s interface was cluttered with obscure language such as “elision” and “AST.” Many programmers found these terms unfamiliar and intimidating.

8.1.4 The Streamlined Scenarios Principle

Common patterns of work should be recognized and then streamlined to avoid awkward interactions such as repetitive clicking or keyboard entry [15, pp. 293–5]. Repetition leads to fatigue, mistakes, wasted time, and forgetting of key information. However, streamlining should not increase the number of ways to carry out a task, which can confuse the user. In the programmer study, programmers had to select a node and perform two pull down selections to trim an arm, a relatively common activity.

8.2 Unifying Design Rules

Although these principles were effective in developing a critique of CStar’s user interface, few ideas for solving these problems emerged. However, when we changed the window-closing functionality to be a button in the upper right-hand corner of every window and gave it the standard label Dismiss, we realized that even this simple change involved the complementary interaction of all four principles: the functionality is now always visible in a button, in a consistent location, with a clear, concise, and consistent name (for instance, “dismiss” does not have the fearful connotation of program termination or costly data loss). Moreover, scenarios involving the closing of windows are streamlined because the button’s location is consistent and it does not require navigation through a menu.

Identifying a user model for the application [20] can facilitate an integrated application of these concepts. A user model represents the way that a user conceptualizes an application, typically through the problem the user is attempting to solve. An interface based on a user model provides a paradigm for accessing functionality and information that is natural to the user and the problem being solved. In contrast, an engineering model represents the way that a system engineer conceptualizes and realizes an application. An application interface based on an engineering model provides direct access to the functionality inside a tool, which makes it ideal for testing and other tool construction activities. However, it is ill-suited to those who do not understand the detailed workings of the tool, and its flexibility can allow uses of the tool that are not efficacious.

Observing that CStar was initially developed under an engineering model, it was apparent that the interface required an overhaul and not just a repair of the particular problems we had identified. Rather than develop a descriptive user model to guide the redesign, however, we chose to develop an operational definition in terms of application-specific design rules that integrated the four HCI principles described above.

8.2.1 The Contextual Visibility Rule

Tool functionality must be directly visible and placed with the data on which it operates. There are several immediate implications of this rule that can be applied as auxiliary rules. One, all functionality must be accessible directly through buttons or item selections, meaning that there

5. A scenario is an account of a sequence of events involving people and artifacts that perhaps has been generalized or simplified to expose important issues [15, p. 81].
should be no pop-up menus or dialogs. Two, pop-up dialogs must be tiled into the window on which the dialog operates or otherwise incorporated into its parent window. This rule addresses several visibility problems and also makes functionality easier to understand because operations can be defined concisely in terms of the surrounding context. At the same time, the number of windows that a programmer must manage are reduced. However, the buttons and tiling can take up extra screen space, which is a valuable resource in this application. To address this problem, we eliminated functionality that could be adequately provided by other tools. We also found that by locating functionality with its data, button names got shorter and some redundant functionality was eliminated.

8.2.2 The Scenario Organization Rule
Windows and subwindows must be laid out and linked according to common user scenarios. In particular, subwindows of a large window should be tiled left-to-right, top-to-bottom in likely order of access. Moreover, selected items are to be retained across operations, and if an operation creates a new data item, that item becomes a current selection. Since functions are located with their data, each subwindow containing data can have its own current selection. Lastly, buttons are also placed in scenario order, and a button invocation that would cause an error given the current state of the tool is disabled and the button text is grayed.6 Highlighting available functionality in this way can help guide the programmer through viable scenarios without the need for extra error windows popping up to signal failures.

8.2.3 The Window-Stealing Rule
Windows should be preemptively closed in a scenario-oriented fashion. To counter the proliferation of open windows, we felt it would be helpful if the tool automatically reclaimed windows that are no longer in use. However, the tool must be able to reliably detect such windows without taking away windows that the programmer still wants. Our decision was to preserve all windows that are created in a depth-first traversal of windows, such as navigating from a star diagram window to a node’s grep window and then from the grep window to a text window for the first item in the grep list. Thus, a “leaf” window can be easily viewed in the context it was derived—visibility is preserved. However, if the parent of a leaf window opens a new window, then the current leaf window is closed and the new one is put in its place. For example, when a grep window’s item selection is advanced to the next item, it opens a new text window to show the selection in context and closes the window for the previous item. This rule does not apply to star diagram windows, since typically few such windows exist and moving between them is not that common.

8.2.4 Example: The Star Diagram Window
As an example of the first two rules, consider the star diagram window (Fig. 5). All functions except zooming are directly available through buttons, and each subwindow has a set of buttons that operate on the data it displays. Exploiting the buttons’ context, many of their names were shortened without losing clarity. The subwindows are tiled in a scenario presentation: In reading from left-to-right, top-to-bottom, the programmer first encounters the elision window, immediately observing what nodes are excluded from the diagram and is led to make changes as deemed necessary. Following the left-to-right, top-to-bottom ordering, the star diagram itself is next examined, and even the star diagram itself tends to be read left-to-right. Next is the trimmed nodes panel, which the programmer may begin using once the star diagram is better understood. Finally, a status display is placed on the bottom of the window, where it is available if needed without intruding upon the programmer’s concentration.

8.3 Evaluation
Although the redesign went smoothly, the design rules conflicted with each other in a few cases, so one rule had to be chosen over the other. First, tiling the star diagram window resulted in giving considerably less space to the trimmed star arms window. However, it was decided that the partial invisibility of its contents could be tolerated, since the window is viewed as a kind of repository for results. Although resizing the overall window can increase the size of this subwindow, this has the negative effect of dramatically increasing the overall screen space occupied by the window, hurting visibility. Second, the tiling of windows puts the buttons of one subwindow in proximity to other subwindows, creating the possibility for mistakenly associating a button with the wrong window. In particular, tiling the star diagram subwindow and the trimmed star arms subwindow resulted in two Trim Arm buttons appearing in the same overall window. The Trim Arm button in the star diagram subwindow trims a selected arm, whereas the Trim Arm button in the trimmed star arms subwindow trims the arm associated with an entry (See Fig. 5. Selecting the wrong button can result in inexplicable behavior. Third, the use of button-graying does not provide an explanation of why certain operations are prohibited, potentially stumping a programmer who wishes to invoke a grayed out button. Self-activating contextual pop-up windows can help, but such a feature is expensive to design and implement; many commercial applications provide only trivial versions of such aids.

To investigate these issues and expose unanticipated problems, we performed another programmer study with just one pair of graduate students with industrial experience, team D, who were familiar with the star diagram concept but had never used a restructuring tool. The experi-

---

6. In the examples in Section 4 the buttons are typically not greyed to improve readability.

7. It did not concern us that the Zoom button invokes a pop-up, because it is currently a rarely used feature and the amount of zooming is apparent from the star diagram itself. Zooming could be handled directly with increment/decrement buttons if redrawing were fast enough, but directly specifying the amount of desired zooming currently saves considerable running time.

8. Recall that with the introduction of the new layered design, a star diagram initially displays only its root—that is, with maximal elision—in order to shorten initial drawing time. When the elision panel was still a hidden pop-up [33], the display of the nearly empty star diagram confused programmers.
mental set-up was chosen to be similar to the previous study so that the results of the new study could be compared with the previous study’s results. As long as team D exhibits no fundamental differences from the previous teams in their choice of design or planning method, then they can be judged to have a similar conceptualization of the tool. Consequently, any low-level differences in their work habits can be fairly attributed to the changes in the user interface, rather than to differences in the programmers themselves. Besides the interface redesign, the only differences in the set-up were the addition of type-based star diagrams and property-based elision to the tool. Since these features add complexity to the interface, they should not inappropriately bias the results of the experiment. However, without engaging a large number of subjects, precisely ascribing differences in behavior to any one design change is difficult. The conclusions reached here are the best judgments of the authors based on the video and audio data, as well as the post-mortem interview, which was used to clarify issues not fully captured by direct observation.

8.3.1 High-Level Observations

Team D chose a design approach based on retrieving and setting fields of \texttt{Objects} elements. This is roughly the design discussed by team B before they opted for the simpler whole-object operations. Although team D did not explicitly acknowledge that this design would require some kind of refinement to permit moving \texttt{Objects} onto disk, they were aware that this was a naive object-oriented design somewhat necessitated by the time constraints of the study. Team D’s planning method consisted of a familiarization phase, followed by construction of a plan using trimming and annotation. When done with planning, team D began their restructuring but did not complete it due to time limitations. Much more than the previous teams, team D tried a variety of elisions to get the kind of view they wanted. Like the previous teams, they conceptualized the nodes in the star diagram as candidates for abstraction into a function.

The similarity of team D’s design and planning approach to the previous teams’ suggests and that there are no fundamental differences in how the team perceives the new tool, permitting their lower level behavior to be fairly compared.

8.3.2 Problems Encountered

Two issues that had concerned us at the start of the study were identified as problems. Team D complained that the trimmed arms panel was too short. The concern about the two \texttt{Trim Arm} buttons was a problem, but perhaps only because of bugs in the trimming interface that led them to false conclusions about how to use it. Fortunately, they found a work-around to achieve their desired goal. Renaming the \texttt{Trim Arm} button in the trimmed star arms subwindow to something like \texttt{Trim Entry’s Arm} should largely resolve the naming conflict, although at some loss of consistency.

Additionally, team D became confused about how to construct a star diagram. After selecting a reference of the \texttt{Objects} variable and then clicking \texttt{Use Type of Var} to make it a root star diagram root, they expected some feedback from the tool, but got none because the main project window was covered, where the selection is logged. The forwarding of the root selection from a text window to the project window violates our rule of locating functionality with the data, but we have not yet found a reasonable way of resolving this problem. Although, this selection-forwarding feature is retained from the earlier version of the tool, in that version it was also possible to create a star diagram from the text window. We have subsequently limited the effects of this problem by separating the selection of the kind of star diagram (type versus variable) from the selection of the variable. The kind is selected from the project window before the star diagram is displayed.

8.3.3 Successes

In general, we found that team D followed scenarios similar to those in the previous study, and consequently their work was streamlined by the tool’s retention of selections and its tiled, button-oriented layout. In particular, the amount of mouse clicking and window management for a scenario was reduced. Only the star diagram creation scenario was impeded, as already described.

Another success was the tiling of project-wide functionalities into a single project window (See Fig. 4). Team D instinctively visited the project window whenever a question or problem arose. This first occurred when they wondered what \texttt{Use Type of Var} did in the text window, and they found that a star diagram entry had been created. The second occurred when they wanted to save out their results, which is indeed accomplished in the project window. The success of the project window can be attributed to the contextual visibility rule, which brought all the project operations together.

The visibility of functionality facilitated experimentation and more sure problem-solving. Team D’s extended experimentation with elision is especially notable. The previous teams largely ignored elision—which was hidden in a menu selection—even though the star diagram for \texttt{Objects} is large. Likewise, trouble-shooting by team D can be characterized as trying out various selections and buttons or checking the project window. The previous teams’ troubleshooting can be characterized as searching through menus and large numbers of windows.

The programmers did not comment about the tool’s automatic reclamation of windows, so we believe that the window-stealing rule is not detrimental, at least for the activity defined in the study. During the post-mortem interview, the programmers said that button-greying is preferable to error dialogs.

8.3.4 Conclusion

The use of HCI principles on their own provided little constructive guidance in redesigning the user interface of CStar. However, they informed the development of an operational user model in terms of design rules that guided a comprehensive redesign of CStar’s user interface.

The design rules and their application had a significant effect on low-level programmer behavior. Most notably, the follow-up study revealed smoother work patterns than observed in our previous study. However, there remain a few
places in the tool where we have not yet found a way to apply the rules. Due to the nature of the problem domain and earlier design decisions that proved difficult to change, these problems may be difficult to eradicate in the future. Another problem is that button-oriented and tiled interfaces do not always scale well as functionality is added. Indeed, there is a need to shrink down or move the elision panel to make more room for the trimmed arms panel.

9 RELATED WORK

Like many program visualizations, a star diagram graphically highlights relationships of interest for some task while hiding irrelevant details. A star diagram planning tool differs from other structural visualization tools in the purpose of the visualization, the granularity at which the program is analyzed and presented, and the planning tasks that the visualization can facilitate. Previous research provides a general comparison of the star diagram to other structural visualization and manipulation techniques [22], [9]. Here we discuss how two other manipulable structural visualization tools can aid the planning of complex changes and manage scale.

The Rigi reverse engineering environment provides semiautomatic construction of graphical views of a system to assist broad architectural understanding [32]. A view based on entities such as functions, files, classes, or modules is initially constructed by the environment and then augmented by the programmer via grouping, collapsing, and hiding. The additions allow the programmer to describe the system in terms of abstractions not visible in the implementation. Such additions can be used to redevelop documentation or to plan changes that are related to managing these invisible abstractions, although there is no specific support for constructing change plans. Rigi’s visual manipulations can be seen as emulating transformations in the same spirit of CStar’s trimming functionality. However, Rigi operates primarily at the level of functions and files rather than at the level of statements and expressions, limiting the kinds of transformations that can be emulated. In contrast to the star diagram’s selective visualization of relevant low-level computational details, Rigi’s general architectural support obligates it to provide a component-level view of every computation in the system. However, like the star diagram’s elision techniques, hierarchical structure in the visualization can be used to collapse several nodes into a single node to provide more efficient use of space.

The SeeSoft system supports visualizing a property of each line of a program (such as its age) as a colored graphical line, thus allowing a programmer to identify patterns between lines in multiple files by comparing colors [17]. Each file of source code in a program is represented as a long box, and each line of code is represented by a horizontal line of pixels in the box. The color of a line represents a value of a metric measured for that line, such as how recently the line was modified, whether a line is associated with a specific modification, or whether a given line is a member of a program slice [42], [3]. (Later enhancements to SeeSoft condense the view further by displaying one or more measurements per file, rather than per line.) Such visualizations can facilitate the planning of program modifications by highlighting historical or intrinsic structure that would be invisible otherwise. SeeSoft supports selective highlighting of lines falling within a specified range of values, which can be used to represent structure or simple maintenance plans identified by the programmer. SeeSoft’s use of graphics to represent file size, position to represent source-line location, and color to highlight source-line relationships obligates it to provide a (low-resolution) representation of every line in the system. However, analogous to property-based elision in the star diagram, displaying only the lines of chosen colors might provide useful compressed views of a program.

10 CONCLUSION

One way to lower the high cost of software maintenance is to restructure an existing system into a more maintainable form. Restructuring through a star diagram visualization can help a programmer understand and manipulate the overall structure of a program for the purpose of creating new data abstractions, thus localizing future changes and lowering maintenance costs. However, it is costly to provide scalable meaning-preserving transformation support. Moreover, a large system is likely developed in multiple languages, further complicating the development of a comprehensive transformation-based tool. These problems led us to apply the star diagram visualization as a mechanism for assisting programmers in planning and carrying out their restructurings without the use of transformations.

Providing such assistance required coping with shortcomings in our original approach to constructing, displaying, and manipulating star diagrams. A large star diagram can be difficult to assimilate. Moreover, we had observed programmers using restructuring transformations merely to change the star diagram view for the recording of a design idea. We have demonstrated that a combination of techniques can accommodate complexity and better facilitate planning a complex restructuring when using the star diagram. Because these techniques increase user interaction, improvements to the tool’s implementation and user interface were also required.

10.1 Techniques for Scalable Restructuring Support

10.1.1 Emulation-Based Planning Support

Meaning-preserving transformations are abandoned in favor of support for planning restructurings. Planning support comes in the form of emulating restructuring transformations. This emulation changes the star diagram view analogously to transformations, serving as a record of the programmer’s intent to perform the restructuring change later, perhaps with an editor. In particular, when a programmer identifies an abstraction with the star diagram, trimming and annotations can be used to emulate the Extract Function and Move into Interface transformations from the star diagram restructuring tool. When the star diagram view becomes empty, all uses of the data structure have been accounted for and the planning process is complete. Grep-like views of node stacks, navigation to the source code, and viewing trimmed arms in isolation
further facilitate understanding, planning, and the eventual restructuring. This planning functionality can be used not only for assisting manual restructurings, but also for nonrestructuring activities or as the interface for a transformation-based restructuring tool.

The programmer study revealed that programmers can indeed plan their restructurings with star diagrams augmented by the trimming functionality and navigational features. In addition, programmers made the trimming mechanism their own, using it in novel ways and developing individualized strategies for combining program understanding, planning, and restructuring.

10.1.3 Support for Multiple Star Diagram Roots

To accommodate characteristics of complex data structures, we customized the star diagram algorithm to support three different methods for creating a star diagram with a root containing multiple variables and expressions: selection of one or more variables, selection of all variables and expressions of a selected type, and the union of existing star diagrams. This feature allows all data elements of interest to be included in a star diagram, helping to ensure that no aspect of the required change is overlooked. Moreover, new elements can be added to the root as they are discovered by unifying star diagrams together. Interestingly, one of the teams in the programmer study used this feature to analyze the correctness their partial changes in the context of their remaining nodes of interest. Second, a programmer can trim a node (and its associated paths) to remove uninteresting computations from the view, cutting down the height of the star diagram. The programmer study also revealed that node trimming can be used to look at an interesting subdiagram in isolation. Measurements of elision and trimming in Emacs and Omega showed that a star diagram can be scaled down to an adequate level. Such scaling might remove too much information, requiring the elision parameters to be frequently changed to keep the most relevant information visible. Although many tools use trees to visualize information and supply some form of depth elision, it is unclear whether property-based elision is generally applicable. It depends on whether the tree nodes of the visualization can be reliably divided into distinct classes that also correspond to their relative usefulness to the problem being solved by the tool user.

10.2 Accommodating Increased Interaction

10.2.1 Layered Design for Better Performance

Early in CStar’s development it was apparent that the shift from transformational support to planning support could result in more interaction, increasing the performance requirements for certain operations. Architectural analysis revealed that the constraints of its pipeline design inhibited the optimization of star diagram elision, which was performed in two steps by constructing a complete diagram and then deleting the unwanted nodes. Evolving the system to a layered design allowed exploiting the presence of a star diagram’s C++ representation to implement one-step elision with selective insertion.

10.2.2 Applying HCI Methods to Streamline Interaction

Increased interaction also demands a facile user interface. Analyzing the work of programmers using CStar revealed egregious violations of four classic HCI principles: visibility of data, consistency, clear language, and streamlined scenarios. Unable to effectively address the problems using these principles, we operationalized the user model as a set of integrated application-specific design rules. These guided a systematic redesign of CStar’s user interface that replaced its hidden pop-up windows and menus of operations with a tiled, button-oriented interface. Observation of one programming team revealed streamlined work, less searching, and a better sense of where to look for solutions when they became confused.

10.3 Future Work

10.3.1 Support for Multiple Languages

CStar supports multiple languages to the extent that any code that is translated to C can be displayed as C code that has been processed by the C preprocessor, . Consequently, the programmers in the programmer studies referred back to the original source code to read comments and inspect macros that had been removed by , as well as to perform restructuring. The programmers mapped between the displayed C code and the original source files using cues such as file names, variable names, and the relative positioning of code in the respective files. These activities suggest that the current approach is a tedious but passable means of handling a limited class of multiple language programs. We see three complementary approaches for coping more satisfactorily with multiple language programs.

To better handle languages that are translated into a common target language, it is possible to augment CStar’s current approach with automated mapping between each source language and the target language. Languages such as yacc, lex, the C preprocessor, and many home-grown languages translate to C, and their translators insert #line directives into the resulting C code so that the C compiler and tools like CStar can map from the C code back to the original source. This approach would allow CStar to display the original source code complete with macros, comments, and original formatting. The downside of this approach is that it is difficult, although not impossible, to identify individual identifiers and expressions within a line for the purposes of selection and highlighting.

For languages that are entirely independent of each other, one viable approach is to provide a tool that contains multiple lexers and parsers, and constructs AST’s in multiple languages. The star diagram constructor would then be responsible for walking these differing AST’s to construct a multilanguage star diagram. The star diagram tool would have to resolve issues regarding what is similar and what is
different for the purposes of stacking. Issues such as the naming of variables and procedures shared amongst the languages need to be resolved as well.

An approach that we have investigated for handling the complexities and syntactic incompatibilities of the C preprocessor is to extend the C parser to parse macros on a best-effort basis [23]. Success in parsing a macro definition results in its definition and uses appearing in the AST, failure in parsing results in it being inlined as usual. Macros that are successfully parsed would not only appear in source code displays, but also in star diagrams.

10.3.2 Case Study

With the improved performance promised by Tcl/Tk 8.0 and better multilanguage support, a realistic case study of the star diagram planning tool concept is feasible. Such a study would permit not only experimentation on a large, commercial system, but also a comprehensive comparison between CStar and tools like grep. The case study could provide insight on the popular use of the tool for planning and making restructuring changes, as well as on the tool’s speed and support for managing large amounts of information. Such a study could also provide insight into the ability of program restructuring to extend the life of a program implementation whose design is no longer helpful in localizing changes.

10.4 Parting Words

A few observations from this work might provide guidance to those developing other kinds of programming tools. First, providing general-purpose, highly automated source-code analysis and manipulation of an entire system is complicated by the need for scalable performance and multilanguage support. Second, the emulation of code changes on program visualizations might serve as a general paradigm for providing low-cost assistance in planning and performing comprehensive program changes. Third, the design of such a tool will likely have its own scale issues, especially in its interface to the programmer, since the programmer is bearing more responsibility in carrying out the work. Finally, with or without heavy automation, tool capabilities that can assist the recall of design information can be valuable.

ACKNOWLEDGMENTS

We thank the reviewers for their guidance on improving the organization and presentation of this paper. Special thanks goes to Walter Korman for his detailed comments on an earlier draft of this paper. We are especially grateful to the anonymous study subjects for their invaluable contributions to this paper.

This work was supported, in part, by the National Science Foundation under Grants CCR-9211002 and CCR-9508745; a Hellman Faculty Fellowship; University of California MICRO Grants 95-065 and 96-063—with Hughes Aircraft; and the U.S. Air Force Office of Scientific Research Contract No. F4-96-209410424.

REFERENCES


GRISWOLD ET AL.: TOOL SUPPORT FOR PLANNING THE RESTRUCTURING OF DATA ABSTRACTIONS IN LARGE SYSTEMS

557


Robert W. Bowdidge received the BA degree in computer science from the University of California, Berkeley in 1989 and the PhD degree in computer science in 1995 from the University of California, San Diego. Dr. Bowdidge is a research staff member at the IBM Thomas J. Watson Research Center, Hawthorne, New York. His research interests center on improving how programmers maintain programs.

Van B. Nguyen received her BS degree in computer science from California State University, Fullerton in 1993, and her MS degree in computer science from the University of California, San Diego in 1997. Her interests include software engineering, compilers, and networks. She is currently a member of the network management software development team at Nuera Communications Inc., San Diego.

J. David Morgenthaler received an AB degree in geography from the University of California, Berkeley in 1983; an MS degree in applied mathematics from California State University, Hayward in 1989; and a PhD degree in computer science from the University of California, San Diego in 1997. Dr. Morgenthaler is an assistant professor of computer science at the Hong Kong University of Science and Technology. His research interests include static program analysis and tools for software understanding, modification, and refactoring.

Jenny L. Cabaniss received the BA degree in computer science from the University of California, San Diego in 1995 and the MS degree in computer science from the University of California, San Diego in 1997. She is a software engineer at Qualcomm Inc., San Diego. Her research interests include user interface design, databases, and telecommunications.

Morison I. Chen received his BS degree in information and computer science from the University of California, Irvine in 1993 and his MS degree in computer science from the University of California, San Diego in 1996. He is currently a software engineer for TRW, Los Angeles. His work experience includes software development, systems integration, technical customer support, and software maintenance.

Robert W. Bowdidge received the BA degree in computer science from the University of California, Berkeley in 1989 and the PhD degree in computer science in 1995 from the University of California, San Diego. Dr. Bowdidge is a research staff member at the IBM Thomas J. Watson Research Center, Hawthorne, New York. His research interests center on improving how programmers maintain programs.

Van B. Nguyen received her BS degree in computer science from California State University, Fullerton in 1993, and her MS degree in computer science from the University of California, San Diego in 1997. Her interests include software engineering, compilers, and networks. She is currently a member of the network management software development team at Nuera Communications Inc., San Diego.

J. David Morgenthaler received an AB degree in geography from the University of California, Berkeley in 1983; an MS degree in applied mathematics from California State University, Hayward in 1989; and a PhD degree in computer science from the University of California, San Diego in 1997. Dr. Morgenthaler is an assistant professor of computer science at the Hong Kong University of Science and Technology. His research interests include static program analysis and tools for software understanding, modification, and refactoring.

Jenny L. Cabaniss received the BA degree in computer science from the University of California, San Diego in 1995 and the MS degree in computer science from the University of California, San Diego in 1997. She is a software engineer at Qualcomm Inc., San Diego. Her research interests include user interface design, databases, and telecommunications.

Van B. Nguyen received her BS degree in computer science from California State University, Fullerton in 1993, and her MS degree in computer science from the University of California, San Diego in 1997. Her interests include software engineering, compilers, and networks. She is currently a member of the network management software development team at Nuera Communications Inc., San Diego.

J. David Morgenthaler received an AB degree in geography from the University of California, Berkeley in 1983; an MS degree in applied mathematics from California State University, Hayward in 1989; and a PhD degree in computer science from the University of California, San Diego in 1997. Dr. Morgenthaler is an assistant professor of computer science at the Hong Kong University of Science and Technology. His research interests include static program analysis and tools for software understanding, modification, and refactoring.

Jenny L. Cabaniss received the BA degree in computer science from the University of California, San Diego in 1995 and the MS degree in computer science from the University of California, San Diego in 1997. She is a software engineer at Qualcomm Inc., San Diego. Her research interests include user interface design, databases, and telecommunications.

Van B. Nguyen received her BS degree in computer science from California State University, Fullerton in 1993, and her MS degree in computer science from the University of California, San Diego in 1997. Her interests include software engineering, compilers, and networks. She is currently a member of the network management software development team at Nuera Communications Inc., San Diego.

J. David Morgenthaler received an AB degree in geography from the University of California, Berkeley in 1983; an MS degree in applied mathematics from California State University, Hayward in 1989; and a PhD degree in computer science from the University of California, San Diego in 1997. Dr. Morgenthaler is an assistant professor of computer science at the Hong Kong University of Science and Technology. His research interests include static program analysis and tools for software understanding, modification, and refactoring.