Research paper:
Collating Results of Syntactic Searches by Context

Robert W. Bowdidge
IBM T. J. Watson Research Center
P. O. Box 704
Yorktown Heights, NY 10598*

Abstract

Maintenance programming tasks often require answering questions such as "how is this data structure used?", "which operations are performed upon this class?", or "where is this value assigned?" Such tasks are difficult with only `grep`-style file-and-line ordered output because the programmer must collate similar lines mentally, rather than directly perceiving the answer to the query from the search's output.

To better answer questions about how a declaration is used, I implemented a new presentation style for a searching tool where matches are ordered according to the context of the use. By searching for all instances of a given type or of a set of similar data structures, the view can also help a programmer restructure or encapsulate a class by noting all the operations performed upon instances of that class.

This paper introduces the concept of collating syntactic search results according to the context of each use, details the implementation of such a view, demonstrates its implementation in IBM's VisualAge C++ 4.0 programming environment, and presents some initial user feedback.

Keywords: program comprehension, searching tools,restructuring

1 Introduction

Maintaining software often requires asking more about *how* data structures are accessed than *where* data structures are referenced. How is the data structure initialized or changed? Which methods are applied to this class? Which operations are performed most often on the variable? Such queries are important during debugging, restructuring a system, planning enhancements to the system, and understanding the current structure of the system.

Searches in commercial tools do not facilitate the *how* questions. Whether searching using lexical [1, 15] or syntactic information [2, 5, 6, 13, 14], searches for uses of an declaration are generally displayed in a `grep`-like fashion, with each use represented by a line of output listing the containing file name and line number for the use, followed by the matching line of source code. `Grep's` presentation simplifies certain tasks such as noting which files contain uses of a declaration, how many uses occur in each file, which files contain large numbers of uses, and which uses occur in close proximity to others. Such information helps a programmer identify cohesive uses in a single module and note single uses in modules that might suggest poor coupling.

However, the `grep`-like view does not facilitate answering questions relevant to maintenance or restructuring. Identifying the range of manipulations upon a variable, or noting which kinds of manipulations are the most frequent can be particularly difficult with `grep`. A programmer attempting such a search must mentally collate the list of matches in order to identify the uses being manipulated in similar ways. To find all assignments to a variable, the programmer must examine every matching line looking for cases where the variable is on the left hand side of an assignment.

Searching tools could instead arrange matches according to context – the operation being performed on the use, or the syntactic form containing the use. Uses could be arranged under context categories. Such an approach facilitates many maintenance tasks. When restructuring, queries about the range of operations or frequency of types of operations could be directly identified from the list of categories. Categories such as a method access with a large number of accesses could suggest cases where a method should be replaced with several different and more specific methods. When debugging, modification sites of a variable could be identified by examining uses in an assignment or and noting if the address of an object is passed to a function. Absence of a category of operations can also help understanding by suggesting common usage of the data structure. If the `next` method of a data structure is not used in a program, it may

*Author contact information: e-mail:bowdidge@watson.ibm.com, phone (914) 784-6580, fax (914) 784-6576

imply that chains of data structures are unimportant in the particular application.

Observations of programmers also suggest the value of ordering by context [3]. In a restructuring study, programmers used `grep` to ensure they performed a *complete* modification. They visited each use in the order displayed by `grep`, decided what modification was needed at the site, and applied that change. Ordering their changes by location forced them to identify all possible contexts for the use, plan all possible changes, and then remember which change matched which context as they progressed through the code. By arranging uses by context, sites that require similar changes are grouped together. The programmers would only memorize possible changes for the uses in the specific context rather than all contexts, minimizing short-term memory requirements and improving *consistency* of the change.

I propose that searching tools that arrange uses according to context are useful for maintenance and program understanding tasks. I first provide a small example of collating references to uses of a variable or class by context, then discuss the concept in detail. I describe the implementation of sorting uses by context in IBM's C++ development environment. Finally, I examine whether the view is helpful to real users.

2 Examples

To demonstrate the concept of collating uses of a declaration by the context of those uses, I demonstrate its use when modifying the `minicalc` spreadsheet. `minicalc` is a C translation of a Pascal textbook spreadsheet example [11]. Spreadsheet cells are implemented as C structures – simple data structures accessed directly rather than through an abstract interface. The globally accessible fields make maintenance changes more difficult because there are no hints or restrictions about how the data structures should be manipulated.

2.1 Searching for named references to a variable or function

Suppose a programmer exploring the code for `minicalc` wants to change the spreadsheet from having a single currently selected cell to allowing a range to be selected and manipulated together. The programmer knows from comments in the code that the `currcell` data structure contains the currently selected cell, and that `currcell` is a structure containing a row (letter) and column (number) index.

To begin to understand `currcell`, the programmer could use any one of a flock of searching tools, asking that tool to find uses of the variable. That tool returns the matches in the style of `grep`, ordered by file and line, with the text of the matching line shown (Figure 1).

Although this view helps the programmer identify clusters of uses in the same file, identifying the range of manipulations upon `curcell` requires the programmer to collate uses according to operation. For example, identifying assignments to `currcell` requires the programmer to examine every match looking for a use on the left hand side of an assignment. Although the process is simple for this example, such a search would be tedious and error prone if the search returned many matches, or if uses occurred in particularly complex calculations.

Instead, the programmer could use a searching tool that collates matches according to the context of each use (Figure 2). To present these results, the searching tool notes the context of each matching use. (Context generally corresponds to the syntactic or parse tree element directly enclosing the use; the exact details are discussed in Section 3.) The uses are collated by the context, with all uses occurring in a given context listed under that context label.

Such a presentation makes the answer for the *how* queries immediately perceivable from the results of the search. Aspects of `currcell` that are difficult to identify in the file-and-line ordered view can be perceived directly in this view. The programmer can see that there are no matches under an "assignment" category, revealing there are no assignments to `currcell` itself. However, examining the "member access" categories for the name and number fields shows that both are only assigned to initial values. There are no other assignments to `currcell` or its fields, showing that `currcell` is not set directly. Because there is only one entry under the "addressOf" category (and that use only occurs in a call to `DoCellChange`), the programmer suspects that the value of `currcell` is actually changed inside `DoCellChange`. A programmer planning an encapsulation or attempting to understand how the `currcell` data structure is manipulated gains this knowledge directly from the search results.

```
182: static CellId currcell;
1620: printf("%c%d ",currcell.name, currcell.num);
1664: currcell.name = MINROW;
1665: currcell.num = MINCOL;
1667: printf("%c%d ", currcell.name,currcell.num);
1695:   DoCellChange(&currcell);
1699:   DoCellExpr(currcell);
1703:   DoCellLabel(currcell);
```

Figure 1: Searching for uses of the string "currcell", arranged in order found in file.

```
addressOf: 1 use
  DoCellChange(&currcell);
call:DoCellExpr: 1 use
  DoCellExpr(currcell);
call:DoCellLabel: 1 use.
  DoCellLabel(currcell)
member access: name: 3 uses.
  printf("%c%d ",currcell.name,currcell.num);
  currcell.name = MINROW;
  printf("%c%d ",currcell.name,currcell.num);
member access: num: 3 uses.
  printf("%c%d ",currcell.name,currcell.num);
  currcell.num = MINICOL;
  printf("%c%d ",currcell.name,currcell.num);
variable definition: 1 use
  static CELLID currcell;
```

Figure 2: Organizing uses of “currcell” according to the context of the use.

2.2 Searching for all references to instances of a class

Similar techniques can be performed on a class or type name. Because programs manipulate instances of classes rather than the classes themselves, uses of a class name tend to appear in declarative contexts such as the typename for a variable or function parameter, or mentioned by name in operations acting on a class, such as C++'s `new`, `sizeof`, or cast expressions.

Finding explicit uses of a class's name can be important when examining the class hierarchy or determining which classes may depend on other classes, but it is less relevant for restructuring tasks. If a search tool can instead find all uses of all instances of a given type, organizing uses by context can characterize the operations performed on instances of the class. Examining all uses of all instances can simplify maintenance by again allowing the programmer to note the range of behaviors performed upon instances.

For example, the `DEPEND` structure in `minicalc` is a pointer structure used to capture computational dependencies between cells. Suppose the programmer wants to encapsulate `DEPEND` so that the traversal algorithm can be improved. To perform such an encapsulation, the programmer must understand how all uses of DEPEND are manipulated – which methods are called frequently, what style of initialization is needed, and how the traversal algorithm visits computations. Figure 3 shows an example of a search for all uses of all instances of `DEPEND`.

By presenting syntactic information, collating uses by context also provides additional facts not visible in a `grep`-like presentation. Usually, file-and-line approaches fold multiple matches of the same line into a single line of output, disguising the fact that several matches occur. When collating by context, such matches appear multiple times for each relevant match. In the `currcell` example, the references to both the `name` and `num` fields in print statements count as two matches, one for accessing the `name` field and one for accessing `num`. In the `DEPEND` search, the single `Malloc` statement appears three times, once for assigning the new memory to a pointer of type DEPEND, once for the cast, and once for the sizeof.

```
!=: 3 uses
  while (nextdep != NULL) {
  while (!found && current != NULL) {
  while (WITH->dependlist != NULL) {
assign: 14 uses
  WITH->dependlist = current;
  current->next = WITH->dependlist;
  current = cells[dependon.name - MINROW][dependon.num -
  nextdep = cells[LINK->zerocells.name - MINROW]
  WITH->dependlist = current;
  current = current->next;
  nextdep = nextdep->next;
  current = (DEPEND *) Malloc(sizeof(DEPEND));
  [LINK->zerocells.num - FAKENIL].dependlist;
  current = cells[dependon.name - MINROW][dependon.num -
  WITH->dependlist = NULL;
  nextdep = nextdep->next;
  current->next = WITH->dependlist;
  current = current->next;
class definition: 1 use
  typedef struct DEPEND {
cast: 1 use
  current = (DEPEND*) Malloc(sizeof(DEPEND));
member access: dependee: 6 uses
  [nextdep->dependee.num - FAKENIL];
  WITH = &cells[nextdep->dependee.name-MINROW]
  LINK->zerocells = nextdep->dependee;
  current->dependee = currcell;
  if (current->dependee.name == dependon.name &&
  current->dependee.num == dependon.num)
member access: next:3 uses
  current->next = WITH->dependlist;
  current = current->next;
  nextdep = nextdep->next;
sizeof: 1 use
  current = (DEPEND*)Malloc(sizeof(DEPEND));
typedef declaration: 1 use
  } DEPEND;
variable declaration: 2 uses
  struct DEPEND *next;
  DEPEND *dependlist; /* ptr to cells this depends on*/
variable definition: 2 uses
  DEPEND *nextdep;
  DEPEND *current;
```

Figure 3: Collating uses of instances of “DEPEND” data structure according to the context of the use.

3 Detailed Description

Sorting by context is reasonably simple, requiring a query mechanism able to find a set of uses of a declaration, a method for identifying the enclosing “context” of each use, and a method for collating these uses. Of these three tasks, only finding the context of each use is unusual. Determining the context of each use requires a definition of context and an algorithm for calculating context on each use of a searched-for declaration.

syntactic construct in which use occurs	label
mathematical operation	*name of operation*
logical operation	*name of operation*
sizeof, cast	*name of operation*
method or function call	call: *function-name*
assignment	assign
member access	member access: *member-name*
initializer	initializer: *member-name*
declaration	*kind of declaration*
definition	*kind of definition*

Table 1: Labels in Sort by Similarity view for C++

3.1 Definition of "context"

The definition of "context" is chosen based on the intended purpose – helping the programmer recognize how a use of a declaration is referenced or accessed in the program. In all the examples and implementations described here, context is defined to be the immediate syntactic construct in a parsed form of the program. This definition has several advantages. The context mapping can be determined quickly using a parse tree representation. The mapping should be straightforward to any programmer examining the source code. For restructuring and maintenance activities, the definition also means that uses falling into the same category have similar representations in the source code, and should be modifiable in a similar fashion. Using the parse tree ensures that the context category has a direct correspondence with the source code. Finally, every use occurs in some syntactic construct, and therefore every use has a context.

With this definition, the ability to identify context becomes almost trivial if a parse tree is available. For each use of the declaration, the enclosing procedure is parsed, and the parse tree node corresponding to the use found. The enclosing context is usually the parent node of that parse tree node. (Less useful parse tree nodes, such as parentheses expression nodes used by some compilers, can be skipped over.) A textual description of the element or operation serves as the label. Table 1 describes the labels chosen for contexts found in C++ programs.

For example, when examining uses of an integer `simplevar`, the use in the statement `printf(simpleVar)` would be "call: printf" The use in the statement `v1 = simpleVar;` would be "assignment". The use in `v1 = simpleVar + 5;` would be "+".

The context of each use is not always just the name of the syntactic construct. Some contexts may be further identified by details about the construct. Uses in function calls and member accesses are named ("call: DoCellLabel") because the different calls can be thought of as different operations on each use, and are therefore important for distinguishing how the uses are manipulated.

When a use occurs in a declaration or definition, the view names the context as "declaration" rather than narrowing the exact context of the use within ("superclass", or "return type"). There are usually few uses of a name inside declarations and the occurrences provide fewer occasions to restructure, so the finer-grained context is less useful. For example, the superclass's name in a class declaration is considered to be in the context of "class declaration". The use of '`TreeNode` in the declaration `TreeNode *t1 = NULL;` would be considered to be "variable definition" context. `extern TreeNode *t1;` would be considered in a "variable declaration" context.

Although identifying context from a parsed representation is simple, identifying context with only lexical tools could be overly difficult depending on the language. In Lisp, the context is usually the function being applied to the name, and thus only requires moving backwards in the text looking for the applied function. In C++ with operators and functions, the process of identifying the operation immediately being applied to the name is much harder, and probably requires being able to at least partially parse the language. Programmers already use regular expressions or sequences of searches to identify uses in similar contexts in C and C++, but accurately identifying context in all cases is either difficult or impossible with only regular expressions.

Although the proposed definition of "context" is syntactic, other definitions are possible. Some maintenance tasks may be more concerned with how the value of a given variable is manipulated rather than how the variable appears in the containing source code. For such applications, dataflow analysis could be used, with the context representing the operation consuming the value. Using dataflow would better capture how data structures are truly manipulated, and would be particularly useful for identifying potential new functions that would replace common code. However, this definition of "context" would be less useful for general-purpose browsing with `grep`-like presentations because any value could be consumed by multiple operations, greatly enlarging the set of matches displayed.

3.2 Extensions to the collating by context concept

The primary benefit of collating by context is that it can reduce the number of matches that a programmer must examine for tasks focused on specific contexts. A programmer searching for assignments may only need to view ten examples of assignments to a variable, rather than a hundred assorted uses of the variable, reducing a large number of uses to a set that can be easily examined and compared by the programmer. If collating by context does not reduce a larger set of uses to a more manageable size, the programmer may still have problems grasping the results of the search or performing a set of repetitive modifications.

There are a number of approaches for extending the mechanism to further reduce the matches the programmer must examine. First, a particular context category could be subdivided according to the context in which each enclosing context lives. This approach mimics the star diagram visualization's multiple levels of context (described in Section 6.)

Some context labels could be further distinguished by details about the manipulation. Uses on the left hand side and right hand side of an assignment could be considered as very different uses, and thus worth distinguishing as "assign:RHS" and "assign:LHS". For some maintenance tasks, distinguishing between assignments to zero and other assignments can also help identify relevant code.

Finally, information about the source file and line may be relevant to restructuring. A programmer looking at the list of uses in an assignment may notice that all but one appear in the same file. The outlier may represent a design flaw, or indicate a special case. Allowing contexts to be further subdivided by file and line could help programmers note such cases.

4 Implementation

The idea of organizing uses according to context has been implemented in IBM's VisualAge C++ 4.0 compiler and programming environment. The integrated development environment (IDE) syntactic search facility called Find Uses allows all uses of a given declaration to be identified and displayed. The results of a search can be grouped and sorted in three fashions within Find Uses: Sort by File, Sort by Scope, and Sort by Similarity, with the mode chosen via a menu. In all cases, Find Uses performs a syntactic search using parsed representations of the program generated by the compiler. (VisualAge C++'s incremental compilation infrastructure provides tools with cheaply and easily generated parse trees [8].)

The Sort by File style places all uses occurring in the same file into a collapsible subgroup, sorting each group by the line where each use occurs. This sorting style produces output resembling `grep`. The Sort by Scope style groups uses by the scope in which the use occurs. The enclosing scope can be a namespace, a function, a class declaration, or the global namespace. Such a view can be useful when the names of the enclosing scope can provide hints about appropriate or inappropriate coupling. The Sort by Similarity style for Find Uses presents uses grouped by context, with the source code line containing the match placed under the corresponding context. Categories are sorted alphabetically, and matches within each category have no defined order.

Find Uses also has an option for displaying all uses of all instances of a class (as described in Section 2.2). This option, called Show Final Uses, can be turned on and off depending on whether the user needs to see sites where the class name is used, or sites where an instance of the class is used. This option works for sorting by similarity as well as the more run-of-the-mill search by scope or file. (See Figure 5.)

Implementing searches by context inside VisualAge C++ highlighted some some unexpected questions about the algorithm. Certain instances of objects of a given type are not explicit in a C++ program. In calls to member functions, the target object can be referred to using the `this` pointer, or can be implicit if the call is within a member function. Find Uses includes these references in uses of all instances of a class by searching member functions for expressions with the implicit `this` expression.

Also, not every instance of a given type is attached to a named variable. Values returned by a function may be an instance of a given type, but Show Final Uses does not include these in the view unless the value is immediately assigned to a local variable of the type – in which case, the temporary variable would be included in the list of instances to display. This is a weakness of the current implementation. Results of casts would also be easy to add as the parse tree accurately represents casts.

5 Experience with Find Uses

To assess whether Sort by Similarity helps programmers in the field, I surveyed users of VisualAge C++ via e-mail about the Find Uses view.

The survey was mailed to fifty companies that were beta users of the product, and was posted on two USENET newsgroups. The survey asked for the amount of time the person had used VAC++, requested examples where the view had been particularly helpful and particularly misleading, asked which of the sorting styles was most and least useful (and why), and asked about the usefulness of the Show Final Uses option. Questions were phrased to elicit long answers and specific examples in order to get descriptions of the usefulness rather than a quantitative useful/not useful metric. Four people responded to the survey. Each was a programmer with four to eight years of programming experience in industry. Four others responded with feedback (usually about the product as a whole), but did not answer the survey's questions.

The four liked the Find Uses view in general. Two of the four commented that the view was useful for simple searches for a specific name, but less useful for "high-end searches" that they performed with a series of `grep`

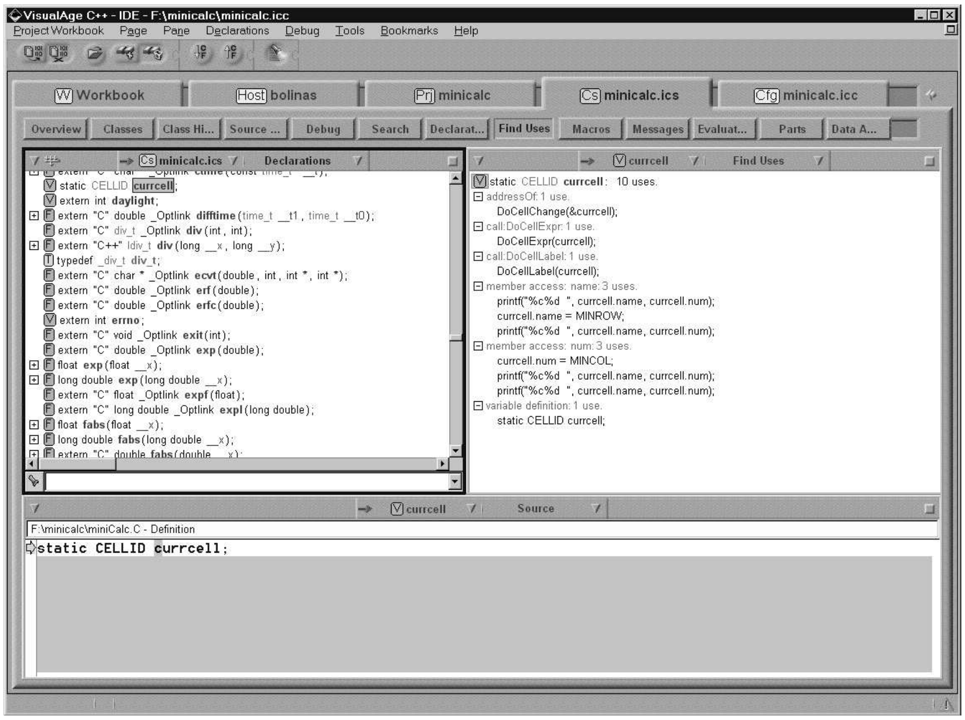

Figure 4: VisualAge C++ 4.0 user interface showing Find Uses in the upper right hand corner of the paned interface. The upper left hand pane shows a list of classes in the user's code, and the bottom pane shows the source code at the location of the selected use.

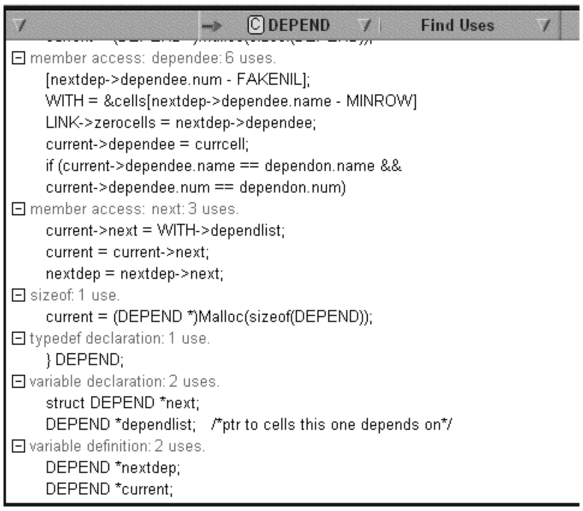

Figure 5: Find uses view in VisualAge C++ 4.0, searching for all uses of all instances of type `Car`.

filters. A third commented that the syntactic search eliminated much of his need to perform multiple searches to improve the accuracy of text searching tools. The fourth said that Find Uses exceeded `grep` in all ways.

Responses for preferred sorting style ranged wildly. One respondent hadn't noticed that the sorting style could be changed, but was happy with Sort by Similarity. The second found Sort by Scope to best match his usual focus, but considered both Sort by Scope and Sort by Similarity to be the most useful. The third felt all three were essential, and the fourth always used Sort by Similarity, "first it was the default and second it makes the most sense – you see how you access the variables and have these accesses grouped together."

For the Show Final Uses option, three had not used the option, and the fourth said "it's nice to have, but that's all."

These users represent a small, self-selecting sample, but their feedback at least suggests that users may be pleased with Sort by Similarity. Some of their comments directly match the intended tasks for the view:

"I had a reference counter for objects I used. I had a bug where objects were deleted too early. Using the Find Uses view I could track down where the reference counter was decremented (and by how much.) I found the problem quite easily."

"It is nice to be able to see all assignments to the variable. Very nice!"

"Helps you in finding out if you use special variables correctly or whether they are needed any more."

Some of their feedback suggests improvements to the view. The pane-style interface did not allow the results of Find Uses to be printed – a major omission. Programmers are used to printing out the results of a search for examination away from the terminal or for ensuring completeness in a large, complex change.

Filtering and sorting uses in a more flexible way was requested, analogous to using multiple filters with `grep`. Such a feature runs counter to Find Uses attempt to provide a familiar and easy to grasp model. Requests for filtering suggest that more flexible syntactic and pattern matching tools [6, 10, 13] are desirable for these users.

6 Related Work

Presentations similar to Sort by Similarity exist in other contexts. In the text domain, the KWIC indexing style [12] or permuted indexes from UNIX manuals display all words in a document in alphabetical order, along with the matching line.

The Sort by Similarity concept is a refinement and simplification of the *star diagram* visualization [4, 7, 9]. (See Figure 6.) A star diagram graphically presents all the operations, direct and indirect, performed on a data structure. The root of the star diagram represents all uses of the declaration being encapsulated. Nodes at the first level of the graph indicate operations directly using the variable. Nodes at the second level represent operations nested around or consuming the result of the previous operation, and so forth. The star diagram groups similar operations scattered throughout the code in the same node. The stacked icon indicates that the node represents more than one similar expression. By selecting a node representing one of these groups, the star diagram user can ask that all the occurrences of the specific set of calculations be turned into calls to a new function manipulating the data structure, where the body of the new function matches the common calculation.

Find Uses can be thought of as a one-level star diagram. The context categories correspond directly with the first level of the star diagram. "Similar operations" in the star diagram are equivalent to similar contexts in Find Uses.

Although the star diagram provides the information needed by programmers for an encapsulation task, its representation may be too task-specific for a general purpose program understanding and maintenance tool. The graph layout primarily helps in selecting an exact set of expressions to turn into a function. The multiple levels in the graph provide added refinement that is less necessary when the programmer is available to scan through a (significantly reduced) set of matches under a category. The star diagram also does not resemble any tool familiar to programmers; overcoming learning curves might seriously affect adoption. For these reasons, the star diagram may be overly complex as a tool for general program understanding and maintenance.

7 Conclusion

Maintenance activities frequently require a programmer to answer questions such as "how is this variable manipulated" or "which methods of the class are most frequently referenced?" Presenting uses of a declaration collated according to the context of each use can solve such problems. This paper presented the concept of sorting by context, proposed one definition for context, demonstrated the practicality of the concept in a commercial product, and showed that initial user feedback is favorable.

Although this work provided some evidence that programmers will use Find Uses and appreciate Sort by Similarity, it would be preferable if first-hand observations showed whether programmers are really using Sort by Similarity for organizing uses by context, or are merely using the grep-like output but not the categories. Observations of programmers using Find Uses while performing general maintenance could test whether Sort by Similarity is truly advantageous.

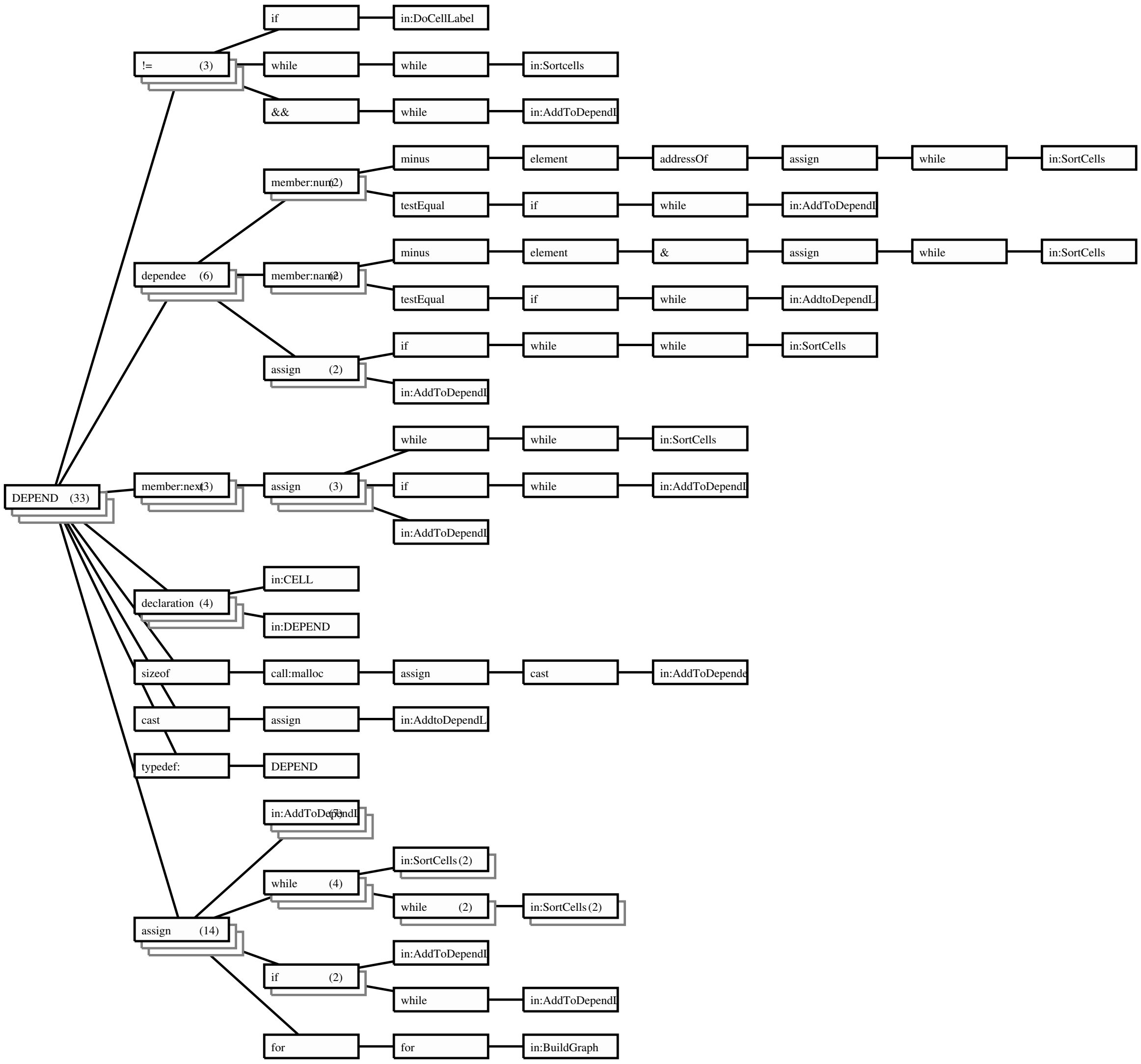

Figure 6: Star diagram for showing similar uses of `DEPEND`. The second level of the graph matches the context categories of the `DEPEND` example.

Acknowledgements

Thanks to Bill Griswold, John Field, and Mark Chu-Carroll for their comments on an earlier draft.

8 Bibliography

References

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

[2] W. R. Bischofberger. Sniff - a pragmatic approach to a C++ programming environment. In *USENIX C++ Technical Conference*, pages 67–81, Portland OR, 1992. USENIX Assoc.

[3] R. W. Bowdidge and W. G. Griswold. How software tools organize programer behavior during the task of data encapsulation. *Empirical Software Engineering*, 2(3):221–268, 1997.

[4] R. W. Bowdidge and W. G. Griswold. Supporting the restructuring of data abstractions through manipulation of a program visualization. *ACM Transactions on Software Engineering and Methodology*, 7(2), 1998.

[5] P. T. Devanbu. Genoa - a customizable, language- and front-end independent code analyzer. In *International Conference on Software Engineering*, pages 307–317, 1992.

[6] W. G. Griswold, D. C. Atkinson, and C. McCurdy. Fast, flexible syntactic pattern matching and processing. In *WPC '96: 4th Workshop on Program Comprehension*, Berlin, Germany, 1996.

[7] W. G. Griswold, M. I. Chen, R. W. Bowdidge, J. L. Cabaniss, V. B. Nguyen, and J. D. Morgenthaler. Tool support for planning the restructuring of data abstractions in large systems. *IEEE Transactions on Software Engineering*, 24:534–558, 1997.

[8] M. Karasick. The architecture of Montana: An open and extensible programming environment with an incremental C++ compiler. In *Sixth International Symposium on the Foundations of Software Engineering*, pages 131–142, 1998.

[9] W. Korman and W. G. Griswold. Elbereth: Tool support for refactoring Java programs. Technical report, University of California, San Diego Department of Computer Science and Engineering, May 1998 1998.

[10] D. A. Ladd and J. C. Ramming. A*: A language for implementing language processors. *IEEE Transactions on Software Engineering*, 21(11):894–901, 1995.

[11] L. H. Miller. *Advanced programming: design and structure using Pascal*. Addison-Wesley, Reading, MA, 1986.

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

[13] S. Paul and A. Prakash. A framework for source code search using program patterns. *IEEE Transactions on Software Engineering*, 20(6):462–475, 1994.

[14] D. S. Rosenblum and A. L. Wolf. Representing semantically analyzed C++ code with Reprise. In *USENIX C++ Conference*, pages 119–134, Washington D.C., 1991.

[15] J. L. Steffen. Interactive examination of a C program with CScope. In *USENIX Winter Conference*, pages 170–175, Dallas, Texas, 1985.