Intertwining Query Construction and Relevance Evaluation

Gerhard Fischer, Scott Henninger, and David Redmiles

Department of Computer Science and Institute of Cognitive Science
University of Colorado
Boulder, Colorado 80309

ABSTRACT

Traditional information access systems generally assume that a well-articulated query exists, and that once an object is found, it can be readily understood. Although this assumption works for retrieving text objects, in more complex domains, such as retrieving software objects for reuse, queries must be incrementally constructed and support is needed for comprehending what is retrieved. Therefore, information access methods need support for query construction and relevance evaluation as an integral part of the location process.

Two prototype systems are described for supporting this need: CODEFINDER for query construction and EXPLAINER for explanations of program examples. These systems interact to support the processes of locating and comprehending software objects for reuse.

KEYWORDS: Information access, software reuse, programming methodologies, cooperative problem solving, retrieval, retrieval by reformulation, explanation, situation model versus system model.

Requirements for Information Access Systems that Support Software Reuse

Traditional information retrieval research assumes that a well-articulated query can be easily thought out and concentrates primarily on retrieval efficiency [3]. Although this assumption works in well-known domains, it does not scale to ill-defined problem domains, in which users need to elaborate and experiment with a problem before the information need is fully identified. Defining the problem is a large part of the problem, and support is needed for an incremental process of exploring the information space while refining the query.

Even in well-known information domains, the problem of query specification can be formidable. Users may know what they are looking for, but lack the knowledge needed to articulate the problem in terms and abstractions that the computer can understand. The underlying problem can be characterized as a mismatch between the terms the system needs, the *system model*, and the *situation model* of the user [9, 13, 24, 26] (see Figure 1). This mismatch exacerbates the vocabulary problem in which it has proven difficult to get an agreed set of terms to describe computer artifacts [17].

In the domain of text objects, the relevancy of information found can easily be judged by users. When retrieving more complex objects, such as software, comprehending these objects becomes a significant problem (see Figure 2). Text objects use a familiar form of language that allows the gap between the situation and system model to be rather small. This is not the case with software objects, in which users may have problems understanding the language, abstractions, and interdependencies upon which the software object is built. This causes the gap between situation and system models to be large enough to require support for judging whether the item meets the information need.

Cooperative problem-solving systems are needed to change the role distribution between humans and computers where currently users construct queries and a system applies queries to its information space [18]. The traditional distribution gives no support to two critical problems: query construction and relevance evaluation. Systems should effectively execute well-specified queries, but they must also support users in articulating what they want and judging the adequacy of objects found.

Information Access in Complex Domains

Software reuse is often touted as the solution to the software engineering crisis [30, 10, 4]. This long-term goal of software reuse is thwarted by an inherent design conflict: to be useful, a reuse system must provide many building blocks, but when many building blocks are available, finding and choosing an appropriate one becomes a difficult problem. The large number and diversity of objects increases the chances that an object closely related to a problem task exists. However, finding such an object among so many is difficult.

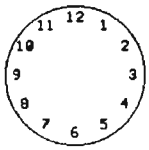
The Information Access Cycle. The process of cooperative problem formation in software reuse intimately intertwines the processes of location and comprehension (see Figure 3). Once an item is found, users may not be able to assess its relevance. The software object must be studied to determine not only what it does and how it is used, but how difficult it will be to modify to the current needs. Once an understanding is achieved, the user is in a better position to understand what his needs are, and can make another attempt at locating an object.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
© 1991 ACM 0-89791-383-3/91/0004/0055...$1.50

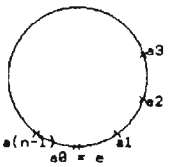
Situation Models		System Models
– ring – doughnut – tire – wheel – washer		- (GRAPHICS:DRAW-CIRCLE x-center y-center radius :INNER-RADIUS i-radius) - CALL BLCIR (xcntr,ycntr,radius) CALL SHADE (xcrds,ycrds,npts,angle,gaps,ngaps,0,0)

An informal study we conducted revealed that people conceptualize the problem of drawing the object shown with one of the situation models indicated. Indexing for situation models corresponds to *application goals*. If drawing a car, drawing a *tire* would be an example of an application goal. The two system models ("inner-radius:" option to "draw-circle" for the SYMBOLICS LISP Machine, and blanking out a circular region before shading a circular curve for DISSPLA) show how system models are indexed by *implementation units*.

Situation Model



System Model



The system model can transcend individual functions. In another study, we observed how people adapted a complete program example from the system to implement a new task in their situation model. Illustrated here is how one subject adapted a diagram of modularity in a cyclic group to drawing a clock face.

Figure 1: Situation Models and System Models for a Graphics Object

Text Object	Software Object
T.A. Standish, *An Essay on Software Reuse,* IEEE Transactions on Software Engineering, Vol. SE-10, No. 5, September 1984, pp. 494-497.	(defun torus (x y r i-r) (graphics:with-room-for-graphics (*standard-output* 100) (graphics:draw-circle x y r :inner-radius i-r)))

Figure 2: Differences in understanding text and software objects

The given literature reference (Text Object) can be understood directly without any specialized knowledge, but users unfamiliar with the specifics of the given LISP code (Software Object) will need help to understand whether it meets their needs.

Previous Information Access Studies. Retrieval by reformulation is a method of constructing and refining retrieval cues. It is based on empirical evidence that ideas are not defined precisely to begin with, but are continuously elaborated upon and evaluated until appropriate cues are constructed, and that people think about categories of things in terms of prototypical examples as opposed to formal or abstract attributes [27, 21]. This paradigm has been applied to create a cooperative relationship between users and systems giving users the ability to incrementally improve a query by critiquing the results of previous queries. RABBIT [31] and HELGON [15] demonstrated how the retrieval by reformulation paradigm could be put into practice.

In another study, we have observed problem-solving behavior of people in a large hardware store (more than 350,000 items) known for its knowledgeable salespeople [29]. We observed that customers did not come in with well-articulated queries, but used a number of specification techniques to eventually arrive at a solution. In many

cases, customers brought an example close to what was needed. In addition to giving customers a means to articulate their query, these examples facilitated the problem-solving process because they gave the salesperson a concrete idea of what was being sought and was crucial to the degree of success in achieving the goal. Salespeople, familiar with potential use situations in their departments, were able to communicate to the customer how items could be used.

CODEFINDER: Information Access for Software Objects

CODEFINDER is an extension of HELGON that retrieves software objects (see Figure 4). Empirical studies of HELGON showed that providing natural means of query formation, which takes advantage of the way human memory works, will lead to better retrieval systems [16]. CODEFINDER uses an associative form of spreading activation [25, 2, 7], which is based on a psychological model of human memory [1, 20] to enhance HELGON. The general idea behind spreading activation is to represent items and keywords as network nodes, with links between the nodes representing mutual association. Construction of a query, which consists of network nodes, causes an activation value to be assigned to those nodes. Activation is then allowed to iteratively spread along the links to associated nodes. Each node updates its activation value by summing all of its inputs by a mathematical formula. Nodes with high activation values are considered to be relevant to the query.

An important characteristic of spreading activation is a flexible inferencing mechanism that can reason with incomplete or imprecise information [25, 8]. Objects do not match the query in an all-or-none fashion, but in varying degrees. Requirements placed by the query are interpreted as *soft constraints*. In contrast to matching algorithms, if a query does not include keywords that index a particular object, that object has a chance of being retrieved if nearby associations exist (see Figure 5 and example below).

In Figure 1, we presented an example of how software objects could be accessed by application goals in addition to implementation units. Figure 5 shows how this can be applied in the CODEFINDER architecture. The keyword *ring* is not connected to *draw-ring*, which draws the object shown in Figure 1 on the Symbolics system. If *ring* is given in the query, it will activate the *draw-circle* node, which in turn activates keyword nodes *tire* and *doughnut*. These keywords will then work together to activate the *draw-ring* node. Since retrieval is performed by soft constraints, it compensates for inconsistent indexing and omitted keywords because keywords are dynamically related through the items they index.

An informal study was performed to compare the query construction methods of CODEFINDER and HELGON. Subjects were given a retrieval task and asked to use either HELGON or CODEFINDER to find a software object (although CODEFINDER takes advantage HELGON's query specification methods, subjects were instructed to enter only keywords and spread activation for CODEFINDER queries). Although the experiment was performed on a very small information space, subjects experienced problems using HELGON [16]. Subjects found CODEFINDER's query construction methods to be quite natural and converged on solutions much quicker, both in time and number of iterations, compared with using HELGON. Some subjects failed to find a solution with HELGON, but all succeeded with CODEFINDER.

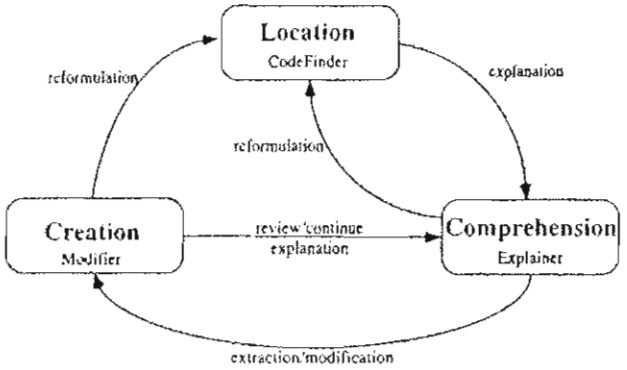


Figure 3: An Information Access Model for Software Reuse

Software reuse involves three cognitive processes: location, comprehension, and modification. The interplay between location and comprehension is discussed in this paper. Support for end-user modifiability in MODIFIER is dealt with in Fischer and Girgensohn [1990].

Network Structure. As shown in Figure 5, there are two structures that compose the network CODEFINDER uses for retrieval. The first is a category hierarchy, which decomposes the information space into a number of abstract categories. This structure is constructed by a domain expert familiar with the information space and able to construct a useful hierarchy. The associative network is constructed automatically by creating links between a software object and its *keywords* and *parameters* attributes (see Figure 4). Keywords can be assigned by the designer of the software object and can be added, deleted, or otherwise modified directly through the CODEFINDER interface.

EXPLAINER: Examples and Explanation to Support Information Access and Modification

Figure 2 illustrates that judging the relevance of software objects was more complicated than retrieval of simple objects such as literature references. Once retrieved, software objects need to be adapted to the task at hand. Both the judgment of relevancy and the adaptation of software objects require the user to understand the retrieved object. The EXPLAINER system supports this need by allowing users to explore the design, implementation, and graphics output of a software object. Figure 6 shows a prototype of the EXPLAINER tool being used to explain the ring example.

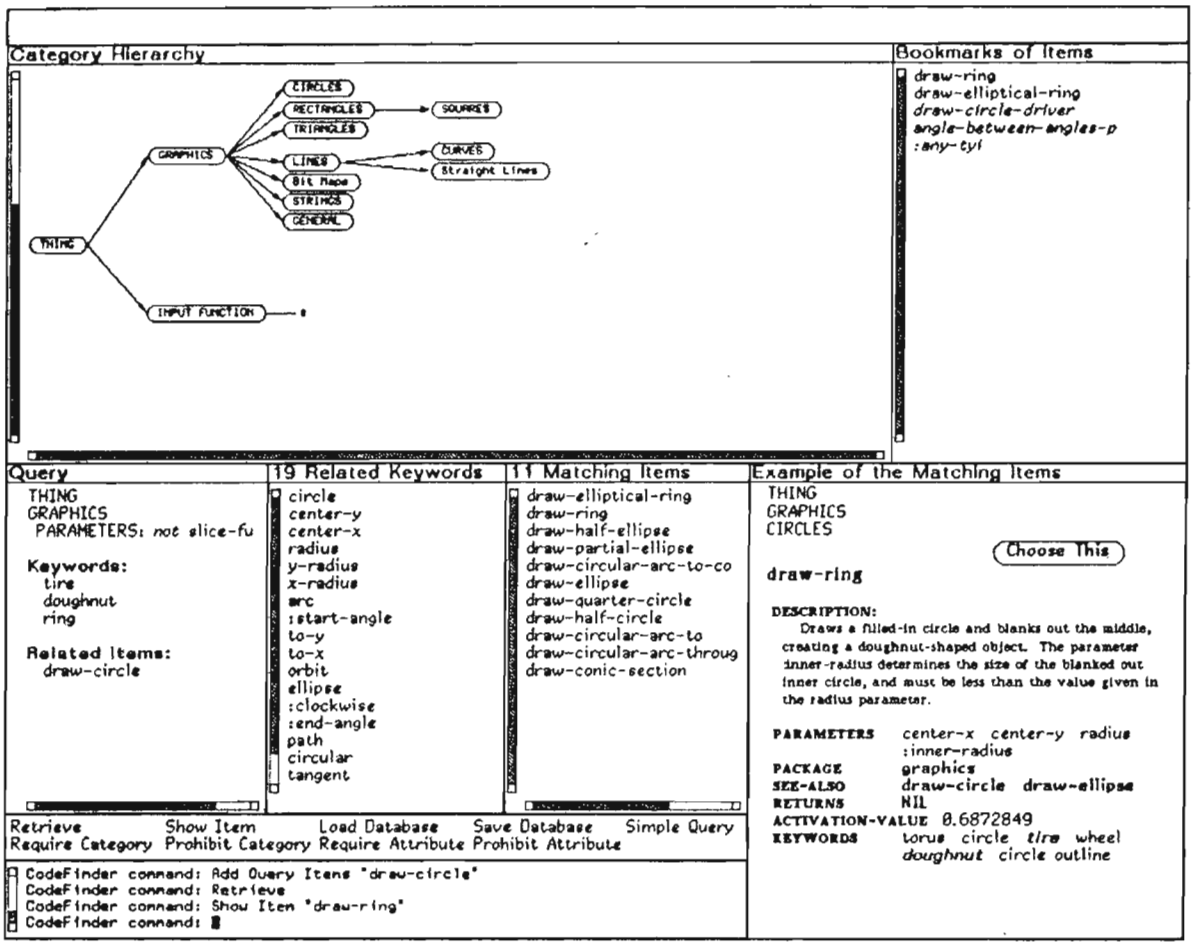


Figure 4: CODEFINDER User Interface

The CODEFINDER user interface is based on HELGON [15]. The **Category Hierarchy** window displays a graphical hierarchy of the information space loaded. In this instance, the information space is a set of graphics functions for the SYMBOLICS LISP Machine. The **Query** pane shows the current query. The top part of the query specifies two categories (*thing* and *graphics*) and a *parameters* attribute. The bottom part specifies keywords and related items. The query parts combine to retrieve the items in the **Matching Items** pane. The **Example of the Matching Items** pane shows the full entry for an item in the information space. The **Choose This** button loads the example item into EXPLAINER for a detailed explanation. The **Bookmarks** pane holds a history of the objects that have appeared in the **Example of the Matching Items** pane. The **Matching Items** pane shows all items matching the current query, by order of relevance to the query. The **Related Keywords** pane shows keywords retrieved by the query. Any of these keywords can be added to the query through mouse action. The remaining panes allow users to specify commands by mouse action or keyboarding (with command completion).

Many studies support the general assumption that examples are helpful in programming [23, 28] and problem solving [6, 29]. A preliminary study we carried out observed specific ways programmers would use examples within our design framework (see Figure 3). Through a description of this study, we discuss below the specific issues of judging relevance, bridging the situation/system model gap, and supporting problem reformulation.

A Preliminary Study. We observed subjects solving simple graphic programming tasks in order to determine what types of questions they would ask about an example and in general what kinds of information they would seek. Subjects were given a programming task to complete and an example from which to start. They were told that the example was related to a potential solution of the programming task, i.e., as if a query and location had already taken place. They worked in an EMACS editor buffer and directed questions and comments to a human; the EXPLAINER prototype had not yet been implemented.

One task was to write a program to draw a clock face (see Figure 1). The subjects were given the example now represented in the EXPLAINER system and shown in Figures 1 and 7. Subjects were observed through questions they asked, spontaneous talking aloud, and mouse cursor movement as they studied different parts of an example's code.

Judging Relevance. When judging the relevance of given

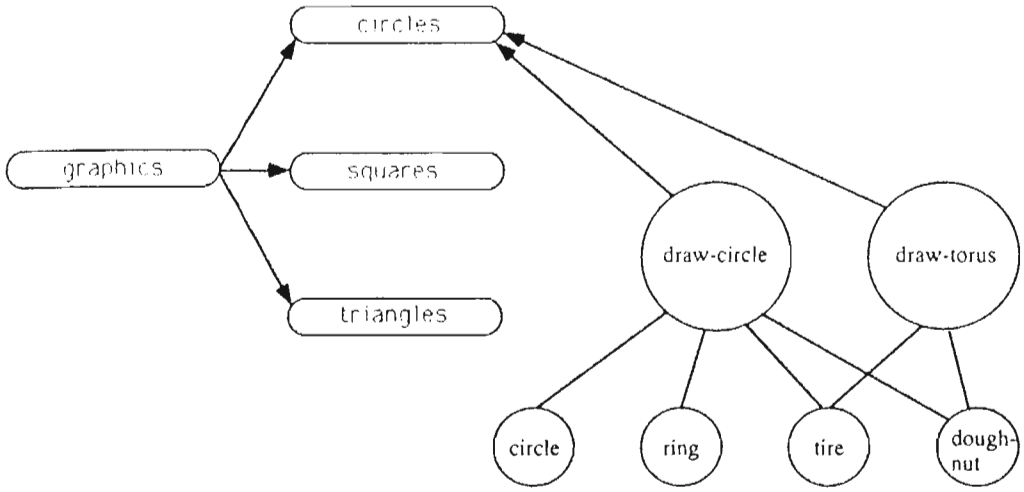


Figure 5: Indexing by Application Goals

The indexing architecture of CODEFINDER makes use of both a hierarchical arrangement of categories and an associative index of keywords. In this figure, ovals represent categories, the smaller circles represent keywords, and larger circles are code objects (keywords and code objects together compose the associative network). The function *draw-circle* is divided into two objects, one represents the function as a whole, and the other represents an option to *draw-circle* (*draw-ring*), which draws a ring. A connection between a keyword and code object means that there is an excitatory association between the keyword and code object. An arrow from a code object to a category means the object is contained within the category.

examples, subjects verified that the example contained certain simple features, such as circle drawing, by inspecting the sample graphics output (see Figure 7). Other features, such as the need to draw numerals, were not so obvious. Subjects studied and/or asked questions about the part of the example code that draws the labels.

Bridging the Situation/System Model Gap. One consideration in using the clock task was that all the subjects were familiar with the problem from their daily experience. What they did not know was how to map the task (their situation model) onto a program in the SYMBOLICS GRAPHICS system (the system model). The example code obviated some aspects of the system model: e.g., the name of the specific functions on the SYMBOLICS for drawing circles and string labels. Explanations clarified the meaning of parameters and the purposes of parts of the code, such as the part that positions the labels along the circumference.

Supporting Problem Reformulation. The interdependency of location (CODEFINDER) and comprehension (EXPLAINER) in our conceptual framework (see Figure 3) occurs when users reformulate their notion of the task and its solution. For example, one subject decided that the numerals on the clock dial should be calculated as integers and then converted to strings for plotting. In the provided example, labels were computed in a different way. The subject asked for an additional example that showed the type of conversion he wanted. This is a relatively simple reformulation, and applies to only a part of the problem solution. Reformulations that replace the initial example entirely are expected when both the CODEFINDER and EXPLAINER systems operate in more realistic settings.

Discussion

To build information access systems that are both useful and usable [11], systems must be more than a passive repository of information. Some of the burden of query construction and relevance assessment must be shifted to the system by supplying knowledge in the world [26] that supports the cognitive tasks of location and comprehension. Knowledge in the world can complement knowledge in the head, allowing designers to concentrate on the creative aspects of a design [12], leaving the more mundane aspects of the task to the system [18].

CODEFINDER addresses the problem of location by acknowledging that browsing techniques do not scale up to hundreds and thousands of software objects and that traditional query search methods break down. They break down for two reasons: (1) users are unable to articulate complete queries and (2) in most systems, software objects are indexed under implementation units and not application goals, leading to a mismatch between the situation and system model. CODEFINDER is a starting point toward solving these problems by combining retrieval by reformulation with associative techniques of retrieval.

Retrieval for complex objects such as software demands tools to aid in comprehension as well as location. Users need to *understand* the retrieved object. They need to know *when* to use a component and *what results* they can expect. The EXPLAINER system emphasizes user-directed explanation of example programs to support users' comprehension of software objects. Exploring a related example can lead users to change their design idea, causing them to return to the location process.

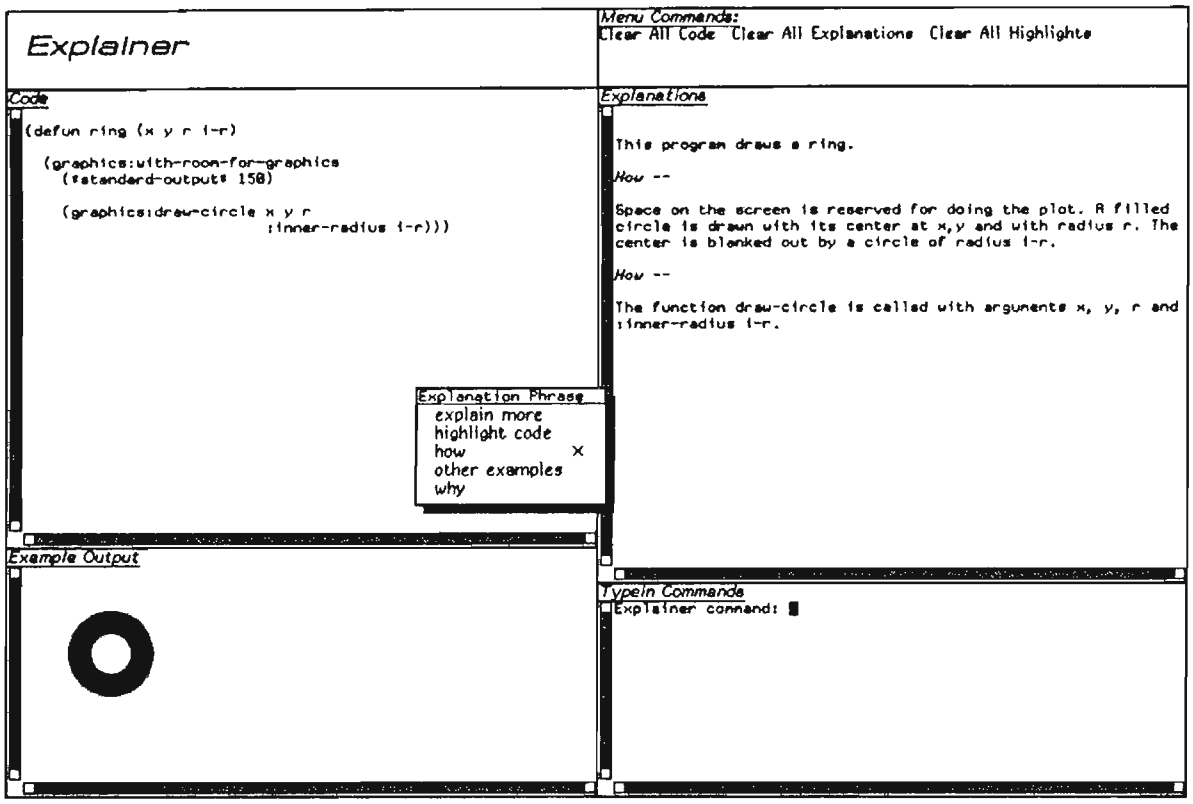


Figure 6: Explaining the Ring Example

Users enter the EXPLAINER system after searching for possible examples in CODEFINDER. For each example, EXPLAINER begins by displaying the code and sample graph on the left half of the screen. An initial description of a few lines of text is displayed on the right. A pane at the bottom allows menu commands to be typed in manually as desired.

The users ask questions, such as "how" or "why", by clicking on a piece of text, code, or graphics. A menu appears, and the users select the question for the selected item. The pop-up menu is shown here in the center of the screen. The constructed question is reflected in the Explanations Pane in italic type. EXPLAINER's response follows. The text shown here reflects a history of a question-answer dialog in progress. The "other examples" menu item allows users to see other possibilities found in CODEFINDER.

Limitations and Future Work. The conceptual framework in which we developed CODEFINDER and EXPLAINER was based on small knowledge bases to test the applicability and functioning of the basic mechanisms. Problems of scaling information access methods to larger information spaces are well documented [5, 19], and we will continue to evaluate our systems on larger information spaces.

In the experiment that led to the development of the scenario, users were allowed to ask questions about both domain and system knowledge. To direct the expansion of the example base, we will test the system against user-defined tasks. Our scenario is based on experiments in which users were assigned graphic programming tasks. Looking at users' own problems not only will provide unbiased guidelines for future development [22], but also will demonstrate how well the gap between situation and system models has been bridged.

Another issue that needs to be empirically explored is the cost-benefit ratio of structure. The value and the cost of a well-structured knowledge base must be compared to the benefit that can be obtained from requiring little a priori structuring. The effort of constructing rich knowledge representations in CODEFINDER and EXPLAINER is offset by what can be gained with better tools in the subdomain of graphics programming.

Conclusions

We have shown that query specification and relevance evaluation must be intertwined in information access systems for software objects. Two operational prototype systems, CODEFINDER and EXPLAINER, are being used to explore and clarify issues involved in this integration. Future evaluations of the systems will continue to improve our understanding of the relationship between query specification and relevance evaluation.

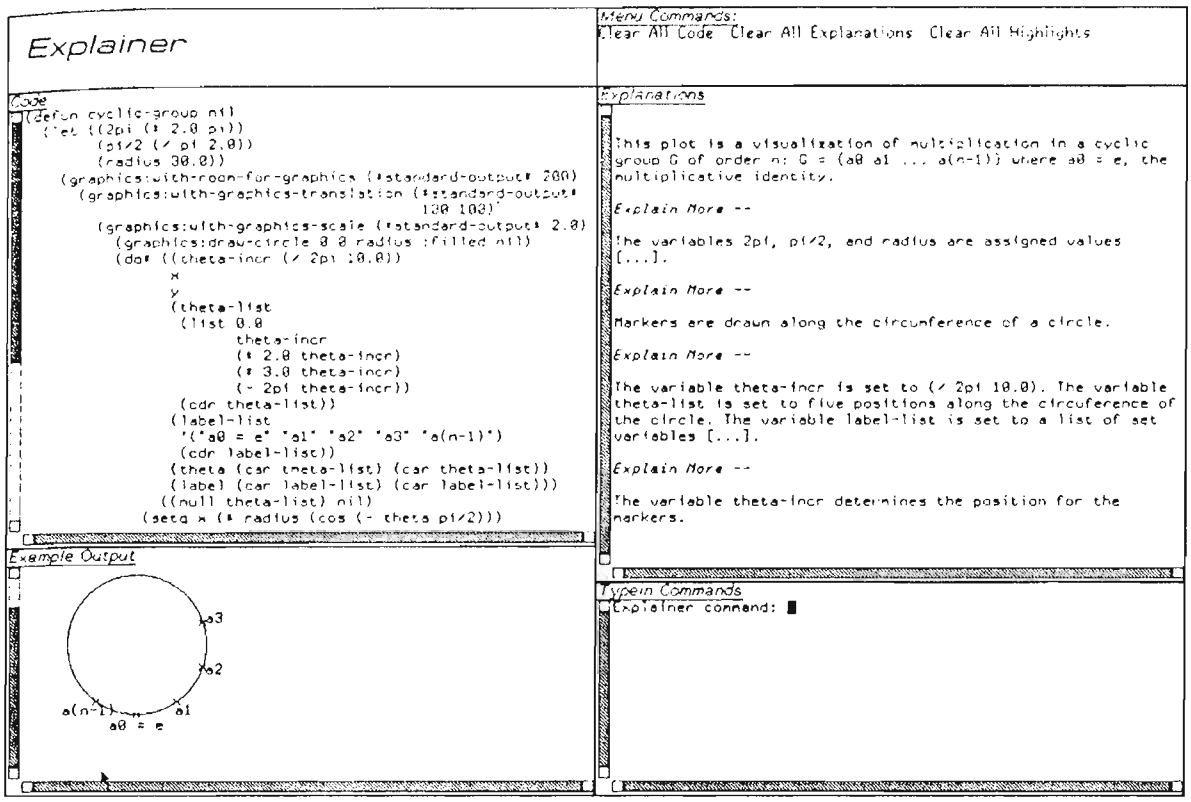


Figure 7: How EXPLAINER Presents the Example Used in the Study

Acknowledgments
We would like to thank the other members of the ARI project, especially Evelyn Ferstl, Peter Foltz, Walter Kintsch, and Curt Stevens with whom we shared many discussions about the ideas and the systems discussed in this paper. The research was supported by Grant No. MDA903-86-C0143 from the Army Research Institute.

REFERENCES

1. J.R. Anderson. *The Architecture of Cognition.* Harvard University Press, Cambridge, MA, 1983.

2. R.K. Belew. *Adaptive Information Retrieval: Machine Learning in Associative Networks.* Technical Report 4, Cognitive Science and Machine Intelligence Laboratory, University of Michigan, 1987.

3. N.J. Belkin, W.B. Croft. Retrieval Techniques. *Annual Review of Information Science and Technology (ARIST) 22* (1987), 109-145.

4. T.J. Biggerstaff, A.J. Perlis (Ed.). *Software Reusability, Volume I: Concepts and Models.* Addison-Wesley Publishing Company, Reading, MA, 1989.

5. D.C. Blair, M.E. Maron. An Evaluation of Retrieval Effectiveness for a Full-Text Document-Retrieval System. *Communications of the ACM 28* (March 1985), 289-299.

6. M.T.H. Chi, M. Bassok, M.W. Lewis, P. Reimann, R. Glaser. Self-Explanations: How Students Study and Use Examples in Learning to Solve Problems. *Cognitive Science 13*, 2 (1989), 145-182.

7. P.R. Cohen, R. Kjeldsen. Information Retrieval by Constrained Spreading Activation in Semantic Networks. *Information Processing and Management 23*, 4 (1987), 255-268.

8. W.B. Croft, T.J. Lucia, J. Cringean, P. Willett. Retrieving Documents by Plausible Inference: An Experimental Study. *Information Processing and Management 25*, 6 (1989), 599-614.

9. T.A. van Dijk, W. Kintsch. *Strategies of Discourse Comprehension.* Academic Press, New York, 1983.

10. G. Fischer. Cognitive View of Reuse and Redesign. *IEEE Software, Special Issue on Reusability 4*, 4 (July 1987), 60-72.

11. G. Fischer. Making Computers more Useful and more Usable. *Proceedings of the 2nd International Conference on Human-Computer Interaction (Honolulu, Hawaii)*, Elsevier Science Publishers, New York, August, 1987, pp. 97-104.

12. G. Fischer. Creativity Enhancing Design Environments. *Proceedings of the International Conference 'Modelling Creativity and Knowledge-Based Creative Design' (Heron Island, Australia)*, October, 1989, pp. 127-132.

13. G. Fischer, P.W. Foltz, W. Kintsch, H. Nieper-Lemke, C. Stevens. *Personal Information Systems and Models of Human Memory*. Department of Computer Science, University of Colorado, Boulder, CO, 1989.

14. G. Fischer, A. Girgensohn. End-User Modifiability in Design Environments. *Human Factors in Computing Systems, CHI'90 Conference Proceedings (Seattle, WA)*, ACM, New York, April, 1990, pp. 183-191.

15. G. Fischer, H. Nieper-Lemke. HELGON: Extending the Retrieval by Reformulation Paradigm. *Human Factors in Computing Systems, CHI'89 Conference Proceedings (Austin, TX)*, ACM, New York, May, 1989, pp. 357-362.

16. P.W. Foltz, W. Kintsch. An Empirical Study of Retrieval by Reformulation on HELGON. In A.A. Turner (Ed.), *Mental Models and User-Centered Design, Workshop Report (Breckenridge, CO)*, Institute of Cognitive Science, University of Colorado (Technical Report 88-9), Boulder, CO, 1988, pp. 9-14.

17. G.W. Furnas, T.K. Landauer, L.M. Gomez, S.T. Dumais. The Vocabulary Problem in Human-System Communication. *Communications of the ACM 30*, 11 (November 1987), 964-971.

18. S. Henninger. Defining the Roles of Humans and Computers in Cooperative Problem Solving Systems for Information Retrieval. *Proceedings of the AAAI Spring Symposium Workshop on Knowledge-Based Human Computer Communication*, March, 1990.

19. B.A. Huberman, T. Hogg. Phase Transitions in Artificial Intelligence Systems. *Artificial Intelligence* , 33 (1987), 155-171.

20. W. Kintsch. The Role of Knowledge in Discourse Comprehension: A Construction-Integration Model. *Psychological Review 95* (1988), 163-182.

21. G. Lakoff. *Women, Fire, and Dangerous Things: What Categories Reveal About the Mind.* The University of Chicago Press, Chicago, IL, 1987.

22. J. Lave. *Cognition in Practice.* Cambridge University Press, Cambridge, UK, 1988.

23. C.H. Lewis, G.M. Olson. Can the Principles of Cognition Lower the Barriers of Programming? In G.M. Olson, E. Soloway, S. Sheppard (Eds.), *Empirical Studies of Programmers (Vol. 2)*, Ablex Publishing Corporation, Lawrence Erlbaum Associates, Norwood, NJ - Hillsdale, NJ, 1987.

24. T.P. Moran. Getting into a System: External-Internal Task Mapping Analysis. *Human Factors in Computing Systems, CHI'83 Conference Proceedings (Boston, MA)*, ACM, New York, December, 1983, pp. 45-49.

25. M.C. Mozer. *Inductive Information Retrieval Using Parallel Distributed Computation.* ICS Report 8406, Institute for Cognitive Science, University of California, San Diego, La Jolla, CA, June, 1984.

26. D.A. Norman. *The Psychology of Everyday Things.* Basic Books, New York, 1988.

27. D.A. Norman, D.G. Bobrow. Descriptions: An Intermediate Stage in Memory Retrieval. *Cognitive Psychology 11* (1979), 107-123.

28. P.L. Pirolli, J.R. Anderson. The Role of Learning from Examples in the Acquisition of Recursive Programming Skills. *Canadian Journal of Psychology 39*, 2 (1985), 240-272.

29. B. Reeves. *Finding and Choosing the Right Object in a Large Hardware Store -- An Empirical Study of Cooperative Problem Solving among Humans.* Department of Computer Science, University of Colorado, Boulder, CO, 1990.

30. T.A. Standish. An Essay on Software Reuse. *IEEE Transactions on Software Engineering SE-10*, 5 (September 1984), 494-497.

31. M.D. Williams. What Makes RABBIT Run? *International Journal of Man-Machine Studies 21* (1984), 333-352.