COGNITIVE TOOLS FOR LOCATING AND COMPREHENDING SOFTWARE OBJECTS FOR REUSE

Gerhard Fischer
Scott Henninger
David Redmiles

Reprinted from PROCEEDINGS OF THE 13TH INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING,
Austin, Texas, May 13-16, 1991

COGNITIVE TOOLS FOR LOCATING AND COMPREHENDING SOFTWARE OBJECTS FOR REUSE

Gerhard Fischer, Scott Henninger, and David Redmiles

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

Abstract

Software reuse is the process of using existing components to create new programs. Before components can be reused, they have to be found and understood by the potential re-user. Two prototype systems are described for supporting this need: CODEFINDER for locating software components and EXPLAINER for allowing a user to get explanations of program examples. The problem of software reuse is greater than simply finding, understanding, and modifying programs: the problem must be viewed as a design process. As users develop a solution to a task, their understanding of the task grows and changes. These processes of refinement and evolution, inherent in design in general, must be accommodated in a software reuse framework. The framework is discussed in terms of current theories of problem solving and empirical studies.

Keywords: software reuse, programming methodologies, cooperative problem solving, retrieval by reformulation, retrieval by spreading activation, explanation, situation model versus system model.

1. Introduction

Software reuse is the process of using existing components to create new programs [5, 6, 13, 39]. However reuse and redesign without adequate systems and support tools for finding, comprehending, and modifying information does not reach its potential as one of the most promising design methodologies.

In this paper, we describe a conceptual framework to facilitate software reuse. We will first show that high functionality computer systems by themselves do not provide sufficient support for software reuse. We then describe a conceptual framework for software reuse. Two systems that support this framework, CODEFINDER and EXPLAINER are presented. CODEFINDER addresses issues on information access for software reuse. Support for comprehending software objects is demonstrated with EXPLAINER. A scenario describing how the two systems are used in in a reuse situation is presented. We conclude by showing how these systems fit into the bigger picture of software development environments, address limitations of our systems, and discuss future directions.

2. Reuse Requires more than High Functionality Computer Systems

Reuse is a promising design methodology because complex systems develop faster if they can build on stable subsystems [38] and reuse supports evolutionary design [11]. Systems built with layered architectures support *human problem-domain communication* [16]: increasingly sophisticated layers of domain-oriented abstractions can be created and designers and users are able to communicate with a system in the terms of the problem domain.

The 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. A *high-functionality computing system* [26] supports reuse by providing many software objects that can be called on or adapted. Software objects may include functions (e.g., in LISP), classes (e.g., in object-oriented systems), whole programs, code fragments, or designs. The large number and diversity of objects increases the chances that an object closely related to a problem task exists. However finding such a object among so many is difficult. In a *low-functionality computing system*, objects are easy to find, but the chances of any one fitting a problem closely are low.

In order to support human problem-domain communication and reuse, high-functionality computer systems are

Recommended by: Lloyd Williams

necessary, but they are not sufficient. Empirical studies [10, 35] have identified the following problems:

- users do not have *well-formed* goals and plans,
- users do not know about the *existence* of components,
- users do not know how to *access* components,
- users do not know *when* to use components,
- users do not understand the *results* that components produce for them, and
- users cannot combine, adapt, and modify components according to their *specific* needs.

Our conceptual framework and systems address these problems in the following way. CODEFINDER uses a combination of two innovative retrieval techniques to support retrieval of software objects without complete knowledge of what is needed. The first, retrieval by reformulation [18, 41], allows users to incrementally construct a query. The second, retrieval by spreading activation, goes beyond inflexible matching algorithms (see Section 3.4). The combination of these techniques yields a flexible retrieval mechanism that provides the means to show the user what components exist and how to access them in the absence of well-formed goals and plans. EXPLAINER uses explanations of program examples to help users comprehend software objects. The examples and explanations help users understand when to use components and what possible results the components have. Whole programs are represented as *cases* as in many case-based reasoning systems [1, 36].

Object-oriented programming and design methods provide the ability to build modules that can be reused directly and incrementally refined; however, these languages by themselves do not support the location and understanding of modules. Browsers have been the tool offered as a solution in these environments [7, 17]. While they allow users to view which modules (classes) are available, they push the entire burden of directing or restricting the search and in understanding objects on the user. Our approach is to create a cooperative problem solving relationship between humans and computers where each is delegated the tasks they are best suited for [21]. Thus, browsing is augmented by queries, and understanding is supported by explanations.

3. A Conceptual Framework for Software Reuse

Software designers must be able to locate and understand components before they can be successfully reused. These problems are best approached as a *cooperative problem solving process* [15] between software designers and support systems that contain enough knowledge to assist designers in mapping between their conceptualization of the problem and the abstractions needed for a computer solution.

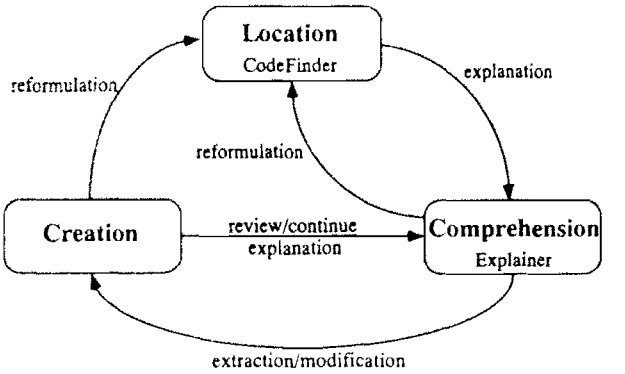

Figure 1: A Software Reuse Model for Examples

3.1. A Model of Software Reuse

Although we are primarily concerned with the processes of location and understanding in this paper, we see these as parts of the larger design process. Our conceptual framework for software reuse uses a model consisting of three phases (see Figure 1): *location* of program examples, *comprehension* of those examples, and *modification* of the examples into a new program. In Figure 1, support could come from either a human consultant or, as we describe here, computer-based systems.

The process begins with locating a software object that is related to the task at hand. In this paper, the general *task* is to create a graphics program. Throughout the paper we will be illustrating concepts in CODEFINDER and EXPLAINER with a task of creating a flattened ring. In Section 4, we provide a scenario showing how CODEFINDER and EXPLAINER support the software reuse cycle.

Locating a software object related to the current task is the purpose of the CODEFINDER system. Once a potential candidate is found, users may not fully understand what the software object does. The purpose of the EXPLAINER system is to answer users' questions with explanations about the object — not only what it does, but how and why as well.

The processes of retrieving and understanding software objects are intimately intertwined. Once a candidate object has been retrieved, users must understand its functionality well enough to decide if the object is indeed

useful for the current task. If not, users must return to the location phase. Location queries need to be incrementally refined using both positive and negative characteristics about the proposed example.

Assuming that users understand what the software object does, chances are that it does not do exactly what they need and therefore a modification process is required. In practice, it is often the case that by integrating a software object into a program, one finds out that the object does not work as anticipated, or it is found to be unsuitable. Thus a re-explanation may be needed, or the initial query may need to be reformulated to look for a different object.

In the cooperative problem solving process (as illustrated in Figure 1), the human users and the computer system each contribute a specific resource or ability [21]. Users contribute their understanding of the task and their ability to draw analogies from the examples to the task [27, 34]. The computer system contributes a repository of examples, along with the means to retrieve and explain them.

3.2. Situation and System Models in Software Reuse

When software designers approach a problem, they often begin at a high level of abstraction, conceptualizing the design in terms of the application problem to be solved [10]. This initial conceptualization must then be translated into terms and abstractions that the computer can understand. The gap between application level and system level in conventional software engineering environments is large. The underlying problem can be characterized as a mismatch between the *system model* provided by the computing environment and the *situation model* of the user [12]. The same problem has been discussed by Moran [30] as external-internal task mapping and by Norman [32] as the gulf of execution and evaluation.

The situation model is an informal and often imprecise representation of what users wish to achieve. It includes some understanding of the task to be done, general design criteria, specific components needed, an acquaintance with related problem solutions, etc.. In order to develop a solution, users must map their situation models into terms the system can deal with.

The following simple example illustrates how software reuse can be facilitated by representing knowledge at the level of the situation model. The traditional approach to indexing software components is to store them by their name. The component's options are perused to find the specific functionality needed. This representation is specific to the system model because it attends solely to the terms that are important to the system (e.g., how it is called, what options are available). For example, if users wish to draw a ring-like figure (as shown in Figure 2) on the SYMBOLICS LISP Machine, they must know the system model which treats this object as the ":inner-radius" option to the *draw-circle* function. They must therefore know to locate this functionality using the name *draw-circle*. Users can locate this information with the help of the SYMBOLICS Document Examiner if they happen to conceptualize the problem like the SYMBOLICS system; the Document Examiner is designed using system terms, knowing nothing about tires or rings.

Support systems must contain enough knowledge to assist users in mapping tasks conceptualized in their situation model to the system model. The initial suggestion by the system may not exactly fit the user's problem. Mismatches may result from terminology [20] or incomplete problem descriptions [25]. Whatever the cause, a cooperative problem solving process between the system and user is needed to attempt to find an adequate solution.

3.3. Retrieval by Reformulation

Retrieval by reformulation is a method of constructing and refining retrieval cues for problems that are not defined precisely to begin with, but are continuously elaborated and evaluated until appropriate information is found [33]. Retrieval can be viewed as an incremental process of retrieval cue construction. This paradigm creates a cooperative relationship between users and a computer system where users are given the ability to incrementally improve a query by critiquing the results of previous queries. The incremental refinement allows the formation of stable intermediate query forms from which users can build upon until the desired results are obtained. RABBIT [41] and HELGON [18] are systems based on the retrieval by reformulation paradigm that have demonstrated how the theory can be put into practice.

CODEFINDER is an extension of HELGON and is designed to retrieve software objects. It provides an interface in which queries can be incrementally constructed through reformulation techniques (see Figure 3). A query consists of categories and corresponding attribute restrictions on the categories. Extensions to HELGON include the ability to create queries with keywords and example database items, as well as a spreading activation process (explained in Section 3.4). The query is specified either through direct manipulation of objects on the screen or through keyboard entry. A crucial feature of the approach is the display of an example item that meets the

Situation Models

- ring
- doughnut
- tire
- wheel
- washer

System Models

- (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 task 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*.

Figure 2: Situation Models and System Models for a Graphics Object

query specification. Not only does this show users an example of what the query is retrieving, but it provides the means to reformulate the query.

Problem Refinement. When retrieving a text object, such as a literature reference in HELGON [18], the object generally will be readily understood. In our terms, the gap between the situation model and the system model is small, making it easy to assess the relevance of a text object. This is not true for software objects, where the gap between the situation and system models can be very large. Support for understanding the object in order to assess its relevance is necessary before the query can be reformulated.

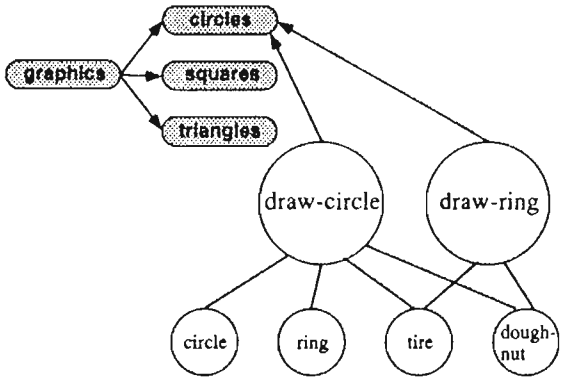

Figure 4: Indexing by Application Goals

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 association between the keyword and code object. An arrow from a code object to a category means the object is contained within the category.

3.4. Accessing Software Objects

CODEFINDER uses associative spreading activation for locating software objects. This technique uses associations to retrieve items that are relevant to a query, but do not exactly match it, thus supporting designers when they cannot fully articulate what they need. Spreading activation is motivated by the hypothesis that providing a means of query formation that takes advantage of the way human memory works will lead to better retrieval systems [19, 21]. It has been used in a number of theories of human memory retrieval [2, 23, 29].

A number of systems have used forms of spreading activation as an information retrieval technique [4, 8, 22, 31]. The general idea is to represent items and keywords as nodes in a network, with links between the nodes representing mutual association. Construction of a query, consisting of network nodes, causes an activation value to be assigned to these 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 and normalizing [31]. Nodes that receive high activation values are relevant to the query.

The spreading activation mechanism in CODEFINDER is

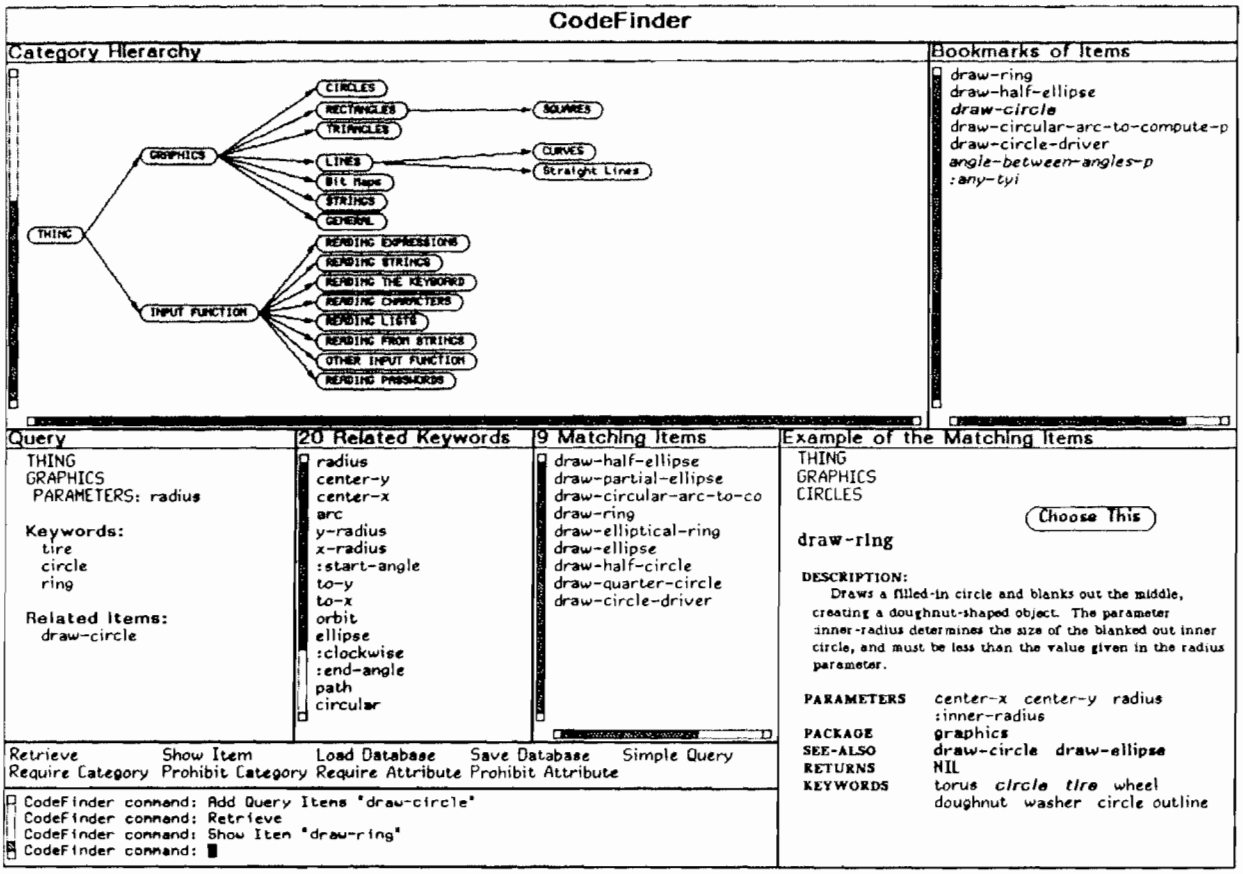

Figure 3: CODEFINDER User Interface

The CODEFINDER user interface is based on HELGON. The **Category Hierarchy** window displays a graphical hierarchy of the information space. The information space used in this screen image 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 one 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. 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. The remaining panes allow users to specify commands by mouse or keyboard.

based on a connectionist version of spreading activation that allows flexible inferencing and reasoning with incomplete or imprecise information [3, 31]. One characteristic of this inductive process is that constraints are satisfied in a flexible manner and can be viewed as *soft constraints*. In most keyword approaches, if a query does not include keywords associated with a particular object, that object will not be retrieved. Since word relationships are constructed dynamically, CODEFINDER is using a unique kind of automatically constructed thesaurus which computes relationships for words in the specific context of the query, avoiding problems of constructing word relationships in the abstract [37].

In our example of drawing a flattened ring, we presented an example of how software objects could be accessed by application goals in addition to implementation units (see Figure 2). Figure 4 shows how this can be applied in the CODEFINDER architecture. The indexing is enhanced by the spreading activation techniques used in CODEFINDER. In Figure 4, the keyword *ring* is not connected to *draw-ring*, which draws the object shown in Figure 2. 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 work together to activate the *draw-ring* node, retrieving the proper object for drawing a ring. As this example shows, "in-

duced'' keywords compensate for inconsistent indexing because keywords are dynamically related through the items they index. These keywords also provide cues for reformulation. By displaying the induced keywords (see Related Keywords pane in Figure 3), users are given an idea of the terminology used the database, minimizing the chance that a query is constructed with keywords that the system does not ''know'' about.

Retrieval by Example. In an empirical study, we observed the problem solving behavior of people in a large department store (over 350,000 items) known for its knowledgeable salespeople [35]. Customers often brought in examples of what was needed. These examples facilitated the problem solving process by giving the salesperson a concrete idea of what was being sought and was crucial in achieving the customers goal.

In CODEFINDER, users are given the opportunity to critique a retrieved example to refine a query (see the Example of the Matching Items pane in Figure 3). The quality of the chosen example plays the same crucial role observed in our empirical observations of human problem solving. The better the example, the easier it is to converge on a satisfactory solution. Therefore, a critical issue in retrieval by reformulation systems is the criteria by which the example is chosen. Previous systems, including HELGON, did not address this issue, but used an arbitrary retrieved item as the example. CODEFINDER enhances the quality of the chosen example by providing a ranking criteria, the activation value, which chooses the item in the knowledge base that is most highly associated with the query.

The empirical study showed that people are often not able to articulate their goals and intentions by words or categories, but explained through examples. CODEFINDER supports such a query specification technique by allowing example items to be included in a query (this is shown in the *Related Items* portion of the query pane). A query that includes example items will retrieve items that are closely associated with it in the same way that spreading activation works with keywords.

Network Structure. As shown in Figure 4, 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 who is familiar with the information space and is able to construct a useful hierarchy. The associative network is constructed automatically by creating links between a software object and its *keyword* and *parameters* attributes (see Figure 3). Keywords can be assigned by the designer of the software object. Keywords can be added, deleted, or otherwise modified directly through the CODEFINDER interface.

3.5. The role of examples and explanation in software reuse

Example-based design of software can play a crucial role in software design and understanding. Lewis and Olson [28] propose that the productivity of casual programmers can be increased by adopting a development style that emphasizes the use of example code. This is based on the hypothesis that ''When grappling with new material, learners often try to adapt examples of earlier material to suit the present situation, modifying the example by analogy'' [28, p. 9]. In another study, a programmer was observed developing code in a familiar object-oriented environment [24]. The study showed that successful reuse of code hinged on the programmer's knowledge of what existed. The programmer made explicit efforts to avoid getting into details of code and instead verified functionality by modifying existing code and observing the results by testing the code.

Anyone attempting reuse is faced with the problems of knowing when to use components, what components do, and how components should be modified. Reuse in high-functionality systems is aggravated because they contain many separate subsystems designed to meet specific needs. The original circumstances for creating these different subsystems and functions can lead to various models of a task. A model of how circle and square drawing routines are organized may have little to do with the model of how those shapes are made sensitive for clicking on by a mouse. Many system models may be involved in programming one task. In cases where there are many different models at work, examples may be the most practical means of explanation. An example can simply state: ''this is how you do that.'' Redundancy is another effect occurring in high-functionality systems: there are many ways of doing one thing. In such environments, examples constrain the possibilities: an example shows *one way* to do something.

Examples provide a bridge between whatever general understanding users have of a software library and the procedural specifics. In the context of the earlier example (see Figure 2), users might know that they need to use a circle drawing routine to draw a ring; an example illustrates how the system uses a parameter of '':inner-radius'' to accomplish this. Programming conventions and heuristics [27] may guide a user's guess, but a picture and corresponding code can show exactly what is needed.

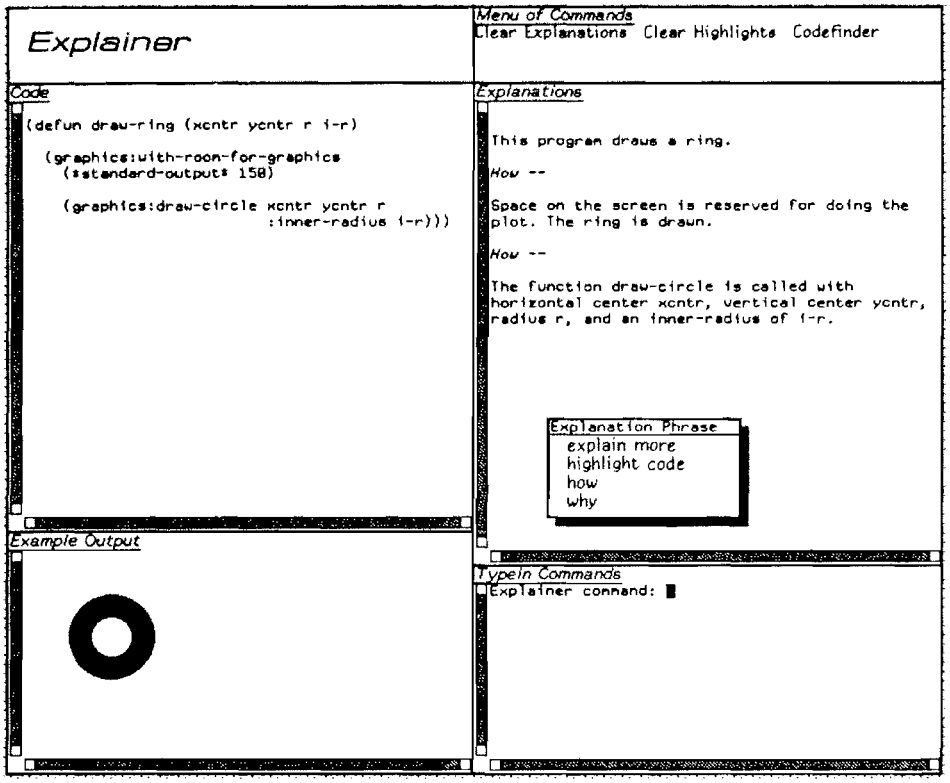

Figure 5: Explaining the Ring Example

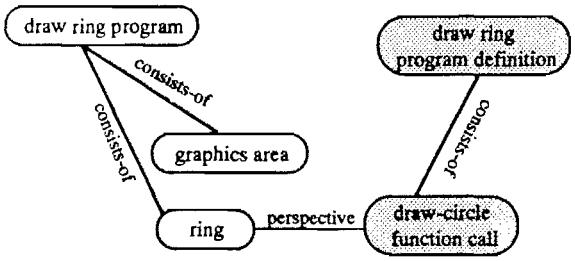

Figure 6: Extract of the Representation for the DRAW-RING Example

Examples are represented by networks of their constituent concepts. Shaded ovals distinguish concepts in the LISP-language perspective from concepts in the graphics perspective. Different views (code, text, graphic — see Figure 5) are generated from this network. In this manner, menu actions on any view are linked to actions on the example as a whole as well as its presentation in other views.

3.6. Explaining Program Examples

Figure 5 shows a prototype of the EXPLAINER tool being used to explain the simple ring example. In the direct manipulation interface, the method of asking questions is through pointing and selection. Segments of the code, features of the sample plot, and phrases in the explanation text are sensitive to the cursor movement.

An example program and the graph it produces are displayed on the left half of the screen. An initial description and subsequent text explanations are displayed in the upper right of the screen. In general, subjects of questions are objects selected from the screen (graphics, text, or code) and questions about these subjects are selected from a menu. The constructed question is reflected in the Explanations Pane in italic type. The lower right pane allows the option of typing commands.

For example, users could click on the initial explanation phrase "this program draws a ring." The menu shown in the middle of the screen appears. From it users could select the question "how" and get a more detailed description. The additional sentences each refer directly to a fragment of code. Selecting either of these sentences and choosing "highlight code" from the menu highlights, the corresponding code fragments.

With each question, the user is directing the explanation provided by the system. The user decides what the next subject will be, what aspect of that subject will be addressed (e.g. how or why), and even the means of the explanation (e.g., more text or highlighted lines of code).

Answers to "how" questions are interpreted through a mapping of concepts (see Figure 6). For example, a plot *consists of* a step to initialize a graphics area on the screen and a step to draw the ring. This explanation gives one *perspective* on the example, a graphics domain perspective. A LISP-language perspective explains that the program consists of a series of specific function calls. *Consists of* and *perspective* are two attributes that define relations between concepts in EXPLAINER's representation of an example program.

Two types of "why" questions are addressed. The simplest is an inverse of "how." The answer traverses the *role in* relation (inverse of *consists of*) to identify, e.g., that the graphics area is initialized as part of the ring program. A second type of "why" question retrieves design rationale, e.g., that the ring is drawn to be in the center of the plotting area. Design rationale link together arbitrary concepts in an examples' concept network.

4. Scenario

Throughout the paper, we have been describing our systems in the context of creating a simple program to draw a flattened ring. As shown in Figure 2, the designer initially conceptualized his task in terms of drawing a ring or tire object. The designer begins the process of locating an example by querying CODEFINDER with the GRAPHICS category shown in the top part of the *Query* pane of Figure 3. The function *angle-between-angles-p* is retrieved (see *Bookmarks of Items* pane). This is not what is being sought, but the description in the *Example of the Matching Items* pane providing some retrieval cues, and the designer reformulates the query by specifying the *radius* parameter in the query. This again does not lead to satisfying results, and the designer selects the *Simple Query* command and enters the keywords *circle, tire,* and *ring*. This retrieves the *draw-circle* function. After inspecting the description of *draw-circle*, the designer decides this is close to what is needed, but lacks the desired feature of specifying the thickness of the line. The designer therefore specifies *draw-circle* to be part of the query, resulting in the query shown in Figure 3.

After performing a retrieval, the designer selects *draw-ring* from the *Matching Items* pane, and decides this function may meet his need. Clicking on the *Choose This* button selects the EXPLAINER system, and loads the *draw-ring* program example (see Figure 5). The designer can explore this example through text, code, and graphic views, as described in Section 3.6. The example describes how to create a ring image but offers no suggestion about flattening out the shape. The user returns to CODEFINDER by selecting the *Codefinder* command from EXPLAINER's main command pane (upper right in Figure 5).

When back in CODEFINDER, the designer refines the query by adding the keyword *oval* (see Figure 8). Evaluating this new query retrieves the function, *draw-elliptical-ring*. The designer again returns to EXPLAINER to review this function (see Figure 7). This example is similar to the previous one, so instead of expanding the initial line of text explanation, the designer proceeds immediately to click on the *draw-ellipse* function call and then select *explain more* from the menu. From the text explanation now given, the designer clicks on the phrase "horizontal semi-axis x-r" to get more explanation. In like manner, the designer proceeds to explore the example until satisfied that this function provides a good basis for his task of drawing a flattened ring.

5. Discussion

Curtis [9] reminds us that reuse is something people have always done: "the hallmark of professionals is their ability to reuse knowledge and experience to perform their tasks ever more efficiently." Simon [38] describes this knowledge as 50,000 chunks requiring as long as ten years to accumulate. Yet professionals do not keep all knowledge in their heads: they rely on knowledge in the world [32].

For software professionals, knowledge in the world includes some understanding of the application domain, some knowledge of the system domain (e.g., libraries of functions, object classes, and both complete and partial program examples), and how the domains are related to each other. Because knowledge in the world can complement knowledge in the head, the human designer can concentrate on the creative aspects of a design [14] and can leave the more mundane aspects of the task to the system [21].

High functionality systems are necessary to extend the knowledge in our head, but they in themselves are not enough. Section 2 lists several problems of high functionality systems. CODEFINDER addresses the problem of location by acknowledging that browsing techniques do not scale up to hundreds and thousands of software objects and traditional query search methods break down. They break down for two reasons: (1) users are unable to articulate complete queries and (2) in most

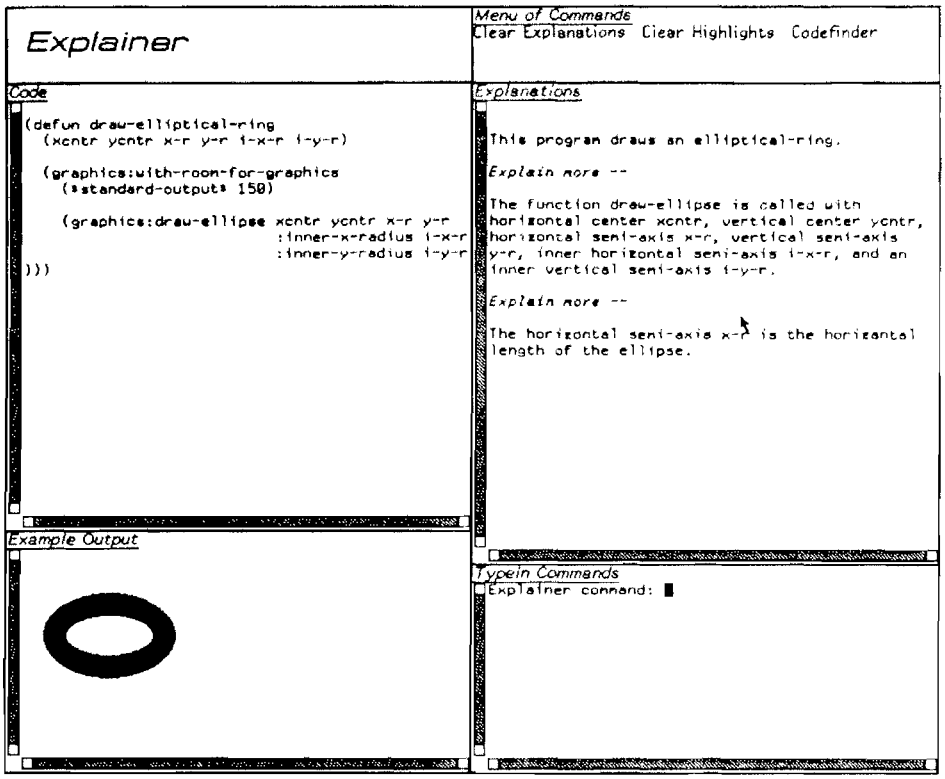

Figure 7: Explaining the Elliptic Ring Example

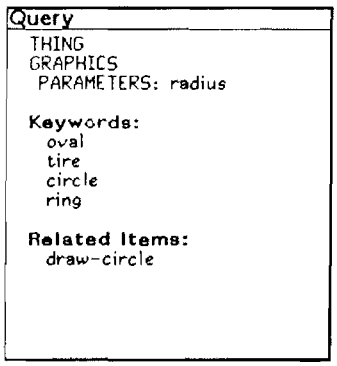

Figure 8: Final Query Pane

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 towards solving these problems by combining retrieval by reformulation with associative techniques of retrieval.

Increasing our chances that we can locate a potentially relevant reuse object does not solve the reuse problem. Users need to *understand* the retrieved object. Users 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 comprehension of software objects. Exploring a related example can lead users to change their idea of their design, causing them to return to the location process.

Limitations and Future Work. Our conceptual framework in which we developed CODEFINDER and EXPLAINER was based on small knowledge bases to test the applicability and the working of the basic mechanisms. Successful reuse requires large information stores to increase the chance to find relevant examples to a great variety of tasks.

In the experiment that led to the development of the scenario, users were allowed to asks questions about both domain and system knowledge. This and follow-up interviews provided insight into what types of knowledge were minimal for reusing software objects. To expand 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 will provide not only unbiased guidelines for future development, but will demonstrate how well the situation to system model gap has been bridged.

Another issue that needs to be empirically explored for software reuse is the cost-benefit ratio. The value and the cost of an experienced professional compared to an entry level employee is well acknowledged and accepted by the software engineering industry — but it is much less clear whether organizations are willing to pay for the design of reusable software systems which will definitely increase the up-front cost of software.

6. Conclusions

An operational software industry must be able to design and build complex software systems. There is ample evidence from other industries that complex artifacts cannot be created from scratch. A marketplace is needed where high-level subsystems and parts are available [40]. Software reuse is not a luxury to turn to: it is a necessity to create the software systems of the future. To gain a better understanding of the problems ahead of us, we have outlined a conceptual framework based on human memory and design research and have tested the relevancy of this framework with two prototype systems for locating and comprehending software objects in a reuse environment.

Acknowledgments

The authors would like to thank the members of the Human-Computer Communication group at the University of Colorado, who contributed to the conceptual framework and the systems discussed in this paper. We would also like to thank the other members of the ARI project, especially Evelyn Ferstl, Peter Foltz, Walter Kintsch, and Curt Stevens. The research was supported by the National Science Foundation under grants No. CDA-8420944, IRI-8722792, and IRI-9015441; by the Army Research Institute under grant No. MDA903-86-C0143, and by grants from the Intelligent Systems Group at NYNEX and from Software Research Associates (SRA) in Tokyo.

References

1. R. Alterman. Adaptive Planning. *Cognitive Science 12*, 3 (1988), 393-421.

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

3. J. Bein, P. Smolenksy. *Application of the Interactive Activation Model to Document Retrieval.* CU-CS-405-88, Department of Computer Science, University of Colorado, Boulder, CO, May, 1988.

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

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

6. T.J. Biggerstaff, A.J. Perlis (Eds.). *Software Reusability, Volume II: Applications and Experience.* Addison-Wesley Publishing Company, Reading, MA, 1989.

7. D.G. Bobrow, I. Goldstein. Representing Design Alternatives. *Proceedings of the AISB Conference*, AISB, Amsterdam, 1980.

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

9. B. Curtis. Cognitive Issues in Reusing Software Artifacts. In T.J. Biggerstaff, A.J. Perlis (Eds.), *Software Reusability; Volume II: Applications and Experience*, ACM Press and Addison-Wesley, Reaing, MA, 1989, pp. 269-287.

10. B. Curtis, H. Krasner, N. Iscoe. A Field Study of the Software Design Process for Large Systems. *Communications of the ACM 31*, 11 (November 1988), 1268-1287.

11. R. Dawkins. *The Blind Watchmaker.* W.W. Norton and Company, New York - London, 1987.

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

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

14. 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.

15. G. Fischer. Communications Requirements for Cooperative Problem Solving Systems. *The International Journal of Information Systems (Special Issue on Knowledge Engineering) 15*, 1 (1990), 21-36.

16. G. Fischer, A.C. Lemke. Construction Kits and Design Environments: Steps Toward Human Problem-Domain Communication. *Human-Computer Interaction 3*, 3 (1988), 179-222.

17. G. Fischer, A.C. Lemke, C. Rathke. From Design to Redesign. In Will Tracz (Ed.), *Tutorial: Software Reuse: Emerging Technology*, IEEE Computer Society, Washington, D.C., 1988, pp. 282-289. Originally appeared in Proceedings of the 9th International Conference on Software Engineering (Monterey, CA).

18. 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.

19. 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.

20. 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.

21. 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, pp. 46-51.

22. W.P. Jones. 'As We May Think'?: Psychological Considerations in the Design of a Personal Filing System. In R. Guindon (Ed.), *Cognitive Science and its Application for Human-Computer Interaction*, Lawrence Erlbaum Associates, Hillsdale, NJ, 1988, Chap. 6, pp. 235-287.

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

24. B.M. Lange, T.G. Moher. Some Strategies of Reuse in an Object-Oriented Programming Environment. *Human Factors in Computing Systems, CHI'89 Conference Proceedings (Austin, TX)*, ACM, New York, May, 1989, pp. 69-73.

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

26. A.C. Lemke. *Design Environments for High-Functionality Computer Systems*. Ph.D. Thesis, Department of Computer Science, University of Colorado, Boulder, CO, July 1989.

27. C. Lewis. Why and how to learn why: analysis-based generalization of procedures. *Cognitive Science 12*, 2 (1988), 211-256.

28. 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.

29. J.L. McClelland, D.E. Rumelhart. An Interactive Activation Model of Context Effects in Letter Perception: Part1: An Account of Basic Findings. *Psychological Review 88*, 5 (1981), 375-407.

30. 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.

31. 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.

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

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

34. 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.

35. 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.

36. C. Riesbeck, R.C. Schank. *Inside Case-Based Reasoning.* Lawrence Erlbaum Associates, Hillsdale, NJ, 1989.

37. G. Salton, M. Smith. On the Application of Syntactic Methodologies in Automatic Text Analysis. *Twelfth Annual International ACMSIGIR Conference on Research and Development in Information Retrieval, SIGIR '89 Conference Proceedings*, ACM, Cambridge, MA, June, 1989, pp. 137-150.

38. H.A. Simon. *The Sciences of the Artificial.* The MIT Press, Cambridge, MA, 1981.

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

40. M.J. Stefik. The Next Knowledge Medium. *AI Magazine 7*, 1 (Spring 1986), 34-46.

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