An Evolutionary Approach to Constructing Effective Software Reuse Repositories

SCOTT HENNINGER
University of Nebraska—Lincoln

Repositories for software reuse are faced with two interrelated problems: (1) acquiring the knowledge to initially construct the repository and (2) modifying the repository to meet the evolving and dynamic needs of software development organizations. Current software repository methods rely heavily on classification, which exacerbates acquisition and evolution problems by requiring costly classification and domain analysis efforts before a repository can be used effectively. This article outlines an approach that avoids these problems by choosing a retrieval method that utilizes minimal repository structure to effectively support the process of finding software components. The approach is demonstrated through a pair of proof-of-concept prototypes: PEEL, a tool to semiautomatically identify reusable components, and CodeFinder, a retrieval system that compensates for the lack of explicit knowledge structures through a spreading activation retrieval process. CodeFinder also allows component representations to be modified while users are searching for information. This mechanism adapts to the changing nature of the information in the repository and incrementally improves the repository while people use it. The combination of these techniques holds potential for designing software repositories that minimize up-front costs, effectively support the search process, and evolve with an organization's changing needs.

Categories and Subject Descriptors: D.2.2 [**Software Engineering**]: Tools and Techniques—*software libraries*; *user interfaces*; H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval—*query formulation*; *retrieval models*

General Terms: Design

Additional Key Words and Phrases: Component repositories, information retrieval, software reuse

1. INTRODUCTION

As libraries of reusable software components continue to grow, the issue of retrieving components from software libraries has captured the attention of the software reuse community [Burton et al. 1987; Devanbu et al. 1991;

This research was supported in part by grant MDA903-86-C0143 from the Army Research Institute.
Author's address: Department of Computer Science and Engineering, 115 Ferguson Hall, CC 0115, University of Nebraska-Lincoln, Lincoln, NE 68588-0115; email: scotth@cse.unl.edu.

Frakes and Gandel 1990; Frakes and Nejmeh 1987; Frakes and Pole 1994; Maarek et al. 1991; Prieto-Díaz and Freeman 1987; Sommerville and Wood 1986]. Especially in component-based reuse approaches, where developers compose software parts into an application, libraries of components are necessary to achieve software reuse. Component-based software reuse faces an inherent dilemma: in order for the approach to be useful, the repository must contain enough components to support developers, but when many examples are available, finding and choosing appropriate ones becomes troublesome. Retrieval techniques as diverse as enumerated classification [Booch 1987], facets [Prieto-Díaz 1985; Prieto-Díaz and Freeman 1987], frame-based classification [Ostertag et al. 1992], free-text indexing [Frakes and Nejmeh 1987], relational databases [Burton et al. 1987], and formal specifications [Chen et al. 1993] have been employed to address the problem of finding relevant components. But issues involving how effective repositories are built, populated, and evolved to meet the changing needs of development organizations have received considerably less attention.

Most retrieval algorithms require that a predefined structure is in place before designers can effectively search the repository. At a minimum, components must be categorized before they are placed in the repository. This value-added approach to building software repositories creates a barrier of real and intellectual capital investments that many organizations cannot overcome [Biggerstaff and Richter 1987]. Most development organizations are pressed to deliver specific products and cannot afford the extra work required to generalize and categorize work products created in individual projects. Methods are needed that can provide adequate retrieval effectiveness with minimal indexing and structuring efforts, allowing organizations to take advantage of valuable assets that have accumulated from previous development efforts without large up-front investments.

In addition to being costly, the structures required for retrieval are often static and unable to adapt to dynamic development contexts. Once the structures are created, they are set in stone, changeable only through complex procedures, often involving redesigning the entire repository. This is a significant problem not only because it has proven exceedingly difficult to get the structure right the first time, but also because the domain is constantly evolving under the pressures of changing technology, project dynamics, and the fluid nature of customer needs.

The purpose of this article is twofold: to describe the problems of indexing and retrieval for software repositories and to present a set of tools designed to support an incremental refinement of component repositories. In our scheme, initial component representations are created with minimal structuring efforts, allowing repositories to be built with minimal up-front effort. Retrieval techniques are then employed to compensate for minimal retrieval structure and to help developers find reusable components. As developers use the repository, adaptive indexing techniques are employed to enhance retrieval structures and evolve the repository toward a representation of the ways its users reason about development problems.

The following sections begin with an overview of current research in retrieval methods for software components. The evolutionary approach to constructing repositories is introduced followed by sections describing the repository seeding, retrieval, and adaptive indexing capabilities of PEEL and CodeFinder, experimental prototypes designed to investigate issues in creating and maintaining software repositories. We conclude with a discussion of future work based on limited empirical investigations of CodeFinder.

2. SUPPORTING SOFTWARE REUSE WITH COMPONENT REPOSITORIES

There are two basic technologies that have been applied in software reuse methodologies: composition-based reuse (also known as parts-based or component-based reuse) and generative techniques (language-based reuse) [Biggerstaff and Richter 1987; Frakes and Pole 1994]. In composition technologies, the components are self-contained entities such as library routines, data structures, programs, objects, and the like. Reusing these components is accomplished by developers finding, understanding, and adapting the components into a new application. Unix pipes and code libraries for programming languages are examples of composition techniques. Generative approaches are closely related to compiler technology, where pieces of parameterized code are invoked by user selection to create customized applications. While library problems can exist for generative technologies [Batory et al. 1994], the problem is more critical for component technologies where enough components must exist for the repository to be useful.

2.1 Component Repositories

The structure of a repository is key to obtaining good retrieval results. No matter how "intelligent" the matching algorithm, if components are indexed or otherwise structured poorly it will be difficult to achieve good retrieval performance. The intuitive and widely held assumption is that up-front investments in structuring a repository result in a proportional increase in the ease with which components can be reused [Barnes and Bollinger 1991, p. 16]. While there is some truth to this assumption, retrieval methods have been created that can effectively use low-cost repository structures. Information retrieval techniques have proliferated over three decades of research [Salton and McGill 1983], but the retrieval methods used for software repositories can be divided into three categories: enumerated classification, faceted, and free-text indexing [Frakes and Gandel 1990]. Hypertext systems have also been used, although some form of retrieval is necessary to make hypertext effective in large repositories [Halasz 1988; Thompson and Croft 1989].

Enumerated classification is a well-known retrieval method used by the Dewey Decimal system and ACM's *Computing Reviews* Classification System. In this method, information is placed in categories that are usually structured in a hierarchy of subcategories, much like the Unix file system.

The appeal of a classification scheme is the ability to iteratively divide an information space into smaller pieces that reduces the amount of information that needs to be perused. The issues involved in using an enumerated classification include its inherent inflexibility and problems with understanding large hierarchies. There is a tradeoff between the depth of a classification hierarchy and the number of category members. Some domains will lend themselves to many small classes. The effect is that users unfamiliar with its structure will become lost in the morass of possible classes [Halasz 1988]. Other domains will have few categories, but must necessarily contain many members. In this case, selection of a class is only a first step in the retrieval process, as the user must then search a large number of category members for relevant information. Another issue is that once the hierarchy is in place, it gives only one view of the repository. Changes to that view may reverberate throughout the taxonomy, resulting in extensive redesign of class structures that can have consequences for the entire contents of the repository.

Enumerated classification requires users to understand the structure and contents of the repository to effectively retrieve information. Even well-thought-out classification systems, such as that used by the Library of Congress, have been shown to be problematic for users [Blackshaw and Fischoff 1988]. In a study with Helgon, a classification-based retrieval system [Fischer and Nieper-Lemke 1989], subjects had trouble distinguishing between class labels such as *Proceedings* and *Inproceedings* in a database of computer science literature references [Foltz and Kintsch 1988]. This led to misunderstandings in the contents of the classes and ineffective retrieval behavior. Part of the problem is that most information retrieval systems assume that an information space can be adequately represented with a single classification. But no one classification is correct under all circumstances, and it is impossible in principle to identify all possible relevant features of a large information space [Furnas et al. 1987]. Even naturally occurring information spaces like biological taxonomies need more than one structure to satisfy the different information needs of researchers in biology and natural sciences [Lakoff 1987]. But these structures have evolved over many years, and the labor-intensive nature of enumerated classification remains a significant barrier to creating multiple structures.

Faceted classification avoids the enumeration of component definitions in a hierarchy by defining attribute classes that can be instantiated with different terms [Prieto-Díaz and Freeman 1987]. This is a variation of the relational model in which terms are grouped into a fixed number of mutually exclusive facets. Users search for components by specifying a term for each of the facets. Within each facet, classification techniques are used to help users choose appropriate terms. This is very similar to the attribute-value structures used in a number of frame-based retrieval techniques in artificial intelligence [Brachman et al. 1991; Devanbu et al. 1991; Patel-Schneider et al. 1984], except that faceted techniques use a fixed

number of facets (attributes) per domain, and no such restriction exists for attribute-value methods [Frakes and Pole 1994].

Facets are more flexible than enumerated schemes because individual facets can be redesigned without impact on other facets. But some of the usability problems remain. While facets make it easy to synthesize and combine terms to represent components, it becomes hard for users to find the right combination of terms that accurately describe the information need, especially in large or complex information spaces [Frakes and Gandel 1990]. The method also requires that users know how the library and terms are structured and have an understanding of the significance of each facet and the terms that are used in the facet [Curtis 1989]. Field use of faceted retrieval systems has shown the need for training people to use facets effectively, and even more extensive training is necessary for designing faceted information domains [Prieto-Díaz 1991].

Free-text indexing (automatic indexing) methods use the text from a document for indexing. Document text is applied to a "stop list" to remove frequently occurring words such as "and" and "the." The remaining text is used as an index to the document. Users specify a query using keywords that are applied to the indices to find matching documents. No classification effort is required, although human indexers are sometimes used to augment automatically extracted index terms. Matching criteria can range from Boolean match to more sophisticated methods, such as the vector model, that use statistical measures to rank retrieved information [Salton and McGill 1983].

Free-text methods are simple to build and retrieve from, but rely on regularities in linguistic texts that need large bodies of text to become statistically accurate. The nonlinguistic nature of source code and the fact that clear and accurate documentation is not necessary for working code make these methods less attractive for software component repositories than for text documents. Free-text methods are most applicable to domains with extensive documentation, such as Unix *man* pages [Frakes and Pole 1994; Maarek and Smadja 1989]. It would be inaccurate to characterize most source code as being documented adequately for these methods. Although retrieval effectiveness of free-text methods has been questioned within text-intensive domains, such as Law [Blair and Maron 1985], the low cost of building the repository coupled with adequate performance has made this approach popular in commercial text retrieval systems and World Wide Web engines such as Yahoo or Alta Vista.

2.2 Repository Structure and Retrieval Effectiveness

These methods define a continuum from enumerated classification that requires extensive structure, to faceted classification requiring less structure, to free-text indexing that requires no structure. Classification schemes are computationally simple to retrieve from (although usability problems exist), but difficult to build. Designers must get the taxonomy "right," and the definition of "right" is elusive, situation-specific, and

dependent on the needs of individuals [Harter 1992]. Faceted classification and attribute-value methods suffer from a reduced form of the same problem—one must get the attributes right; then it becomes a simple matter of choosing terms within the facets. Free-text indexing involves little up-front costs, but is most effective when a large corpus of linguistic text is available.

The strength of methods that use sophisticated information structures is that people can use the knowledge contained in the structures to lead them to relevant information. The weakness is that no support is possible if the information is not structured in a manner consistent with user expectations. Creating specialized information is a labor-intensive and costly process that is prohibitory where resources are strained or where large information spaces are concerned. While it may be true that good structure is crucial in a retrieval system's effectiveness, sophisticated information retrieval techniques have been used that finesse the relationship between repository structure and retrieval methods. The question then becomes the following: what is the relationship between structure and effective retrieval, and can adequate retrieval effectiveness be accomplished with minimally structured repositories?

In what follows, a middle ground is achieved by investigating how much can be accomplished with few assumptions about the existence of specialized knowledge while effectively utilizing information structures that can be cheaply derived. Retrieval methods are employed that need very little structure, yet yield effective retrieval performance. The structures are then augmented in the context of use. As people find information, their interaction with the system is traced and used to improve the existing structure. The result is a flexible integration of retrieval and repository construction methods that improve the underlying structures as the repository is used.

3. CODEFINDER: INFORMATION ACCESS FOR SOFTWARE REUSE

Figure 1 shows how repositories can be constructed with minimal up-front effort and then incrementally improved as the repository is used. In the initial stages of constructing a repository it is important to populate the repository with components, even if the components are not explicitly designed for reuse [Caldiera and Basili 1991, p. 63]. In our approach, a repository is initially "seeded" [Fischer et al. 1994] with a low-cost repository construction method that semiautomatically indexes components with terms and phrases. The prototype system described in this article, called PEEL (**P**arse and **E**xtract **E**macs **L**isp), extracts components from text files and indexes them through a combination of automatic extraction and interactive user support.

The resulting repository, having been created with minimal retrieval structures, may suffer from incomplete and inconsistent indexing, making it difficult for keyword-matching algorithms to retrieve relevant information. A combination of soft-matching and query reformulation techniques is employed to compensate for the minimal effort in designing the repository,

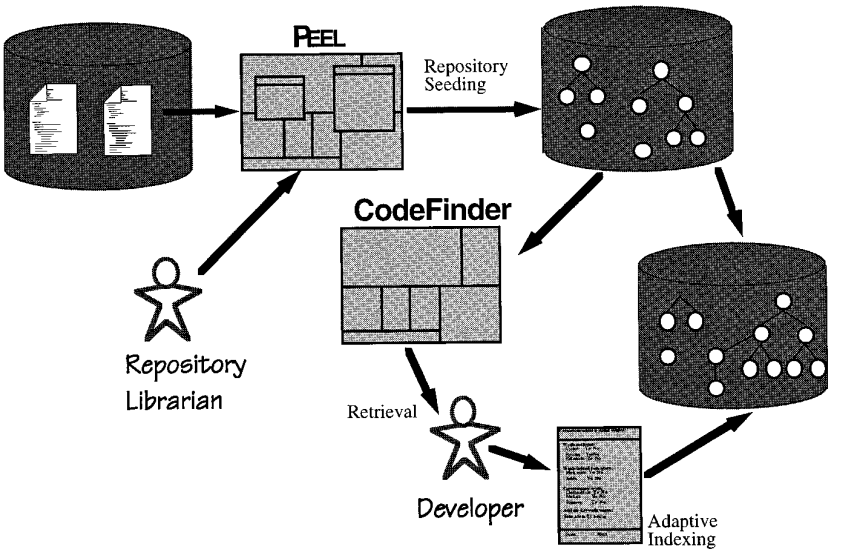

Fig. 1. Evolutionary construction of repositories.

making it easier to find information. The CodeFinder system is a prototype retrieval tool that uses a combination of retrieval techniques to help users find reusable software [Henninger 1994].

Even with effective retrieval tools, a repository's effectiveness is directly proportional to the quality of indexing and structures for retrieval. Relevance feedback and adaptive indexing are two methods that can be used to improve repository structures to reflect the mental models used by developers when searching for components. Empirical studies have shown that evolving network structures can be designed to become more effective over time in a team setting where people have similar concerns and use similar vocabulary [Belew 1987]. In addition, such a system adapts to changes in an organization's software development environment and business needs.

The overall advantage of this approach is that costs are incurred incrementally on an as-needed basis instead of requiring an extensive up-front repository design effort. Initially, components are only required to undergo whatever certification process is necessary to become part of a production system. Subsequent efforts can then incrementally add value to components as they are reused. Techniques for adding value to components have been studied extensively in the literature and include methods such as certification procedures for reusability [Caldiera and Basili 1991] and various domain analysis methods [Prieto-Díaz and Arango 1991; Simos et al. 1995]. The purpose of the work reported here is to build a repository infrastructure that can accommodate these techniques and incrementally add value for the purpose of finding components that meet a developer's task at hand.

To be useful, a repository needs to be supported by (1) tools to create initial information structures, (2) flexible mechanisms to search and

browse the repository, and (3) tools to refine and adapt information as users work with the repository. The combination of CodeFinder and PEEL accomplishes this goal through methods that support semiautomatic extraction of source code components, query construction techniques, and adaptive indexing facilities. Figure 1 shows how these tools work in concert to support the evolution of software repositories.

3.1 The CodeFinder-PEEL Repository

CodeFinder and PEEL were designed to investigate the cost/benefit tradeoffs associated with building and using a repository to support the design process. These issues have been investigated in the domain of Emacs Lisp customizations for the GNU Emacs text editor [Cameron and Rosenblatt 1991; Stallman 1981]. Components are Emacs Lisp functions, variables, and constants that define two email and three network news reader applications that can be executed from the Emacs text editor environment. Emacs Lisp is a variant of Lisp that has specialized constructs and primitives for editing and displaying text on windows. A repository was built with files downloaded from FTP sites and newsgroups using PEEL to extract components and translate them into a CodeFinder representation [Henninger 1993]. Source code was extracted from various newsgroups devoted to Emacs and Emacs LISP issues, including *comp.emacs, gnu.emacs.sources*, and more specialized groups such as *gnu.emacs.gnus* and *gnu.emacs.vm.info*. A modest effort with PEEL resulted in the creation of a repository consisting of more than 1800 Emacs LISP functions, variables, macros, and constants, and about 2900 distinct terms.

4. SEEDING THE REPOSITORY

Seeding a repository is a matter of creating component representations by indexing them with key terms and phrases. Components can take on any size or form, depending on the needs of repository users. PEEL is a reengineering tool that translates Emacs Lisp files into individual, reusable components in a frame-based knowledge representation language named Kandor [Devanbu et al. 1991; Patel-Schneider et al. 1984] that is used by CodeFinder to index components and create a frame-based hierarchy of retrieval concepts [Henninger 1994]. PEEL extracts source code definitions of functions, variables, constants, and macros from a source code file.[1] Information is extracted from each of the components and translated into Kandor objects. Kandor enables the expression of constraints on members of a defined frame. Frame definitions are organized

[1]At this point, the repository only contains source code functions and data structures. Nothing in CodeFinder prevents placing other types of artifacts such as larger-grained components or supporting documentation. While it is desirable to support higher-level reuse, such as subsystems [Wirfs-Brock and Johnson 1990], architectures [Shaw and Garlan 1996], or megaprogramming [Biggerstaff 1992; Wiederhold et al. 1992], the functionality and level of granularity described here are characteristic of current software reuse repositories, which are designed to find and reuse functions and objects to create new programs.

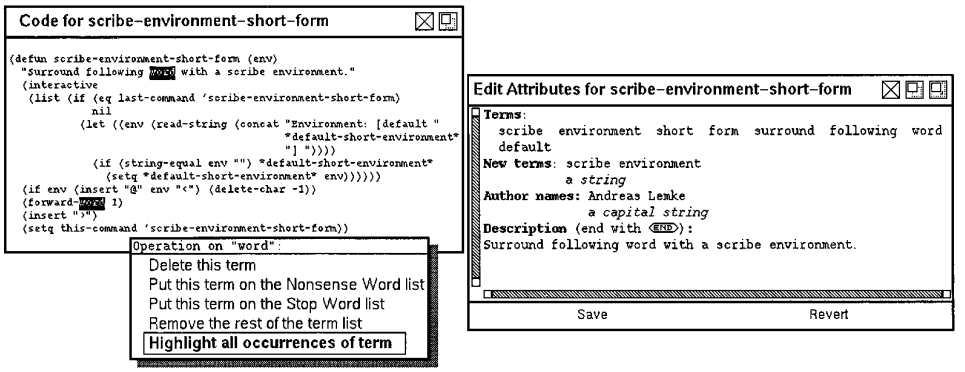

Fig. 2. Highlighting and other indexing operations.

hierarchically and defined through restrictions on superframes. Part of the hierarchy for the Emacs Lisp repository is shown in Figure 4 (shown later). CodeFinder [Henninger 1994], as well as LaSSIE [Devanbu et al. 1991], Argon [Patel-Schneider et al. 1984], and Helgon [Fischer and Nieper-Lemke 1989], has used the inferencing capabilities of inheritance and classification to retrieve information with Kandor representations. Kandor's inferencing capabilities are described in detail elsewhere [Devanbu et al. 1991; Patel-Schneider et al. 1984]. For our purposes, Kandor representations can be viewed as a set of attribute/value slots that contain information about a given component. PEEL is used to capture that information and place it in Kandor slots.

A knowledge representation system is only as good as the knowledge it contains. Automatic extraction is certainly cost-effective, but there are some important slots that need augmenting, such as the TERMS, AUTHORS, and DESCRIPTION slots. As shown in Figure 2, for each component, PEEL will display the source code (window on left-hand side of Figure 2) and allow the user to edit part of the representation (window on right-hand side). Users are free to augment the representation with any terms and phrases deemed useful.

Given the use of nonwords and esoteric abbreviations that are so common in source code, terms are especially important for retrieving software components. For example, one of the mail systems parsed by PEEL uses the abbreviation "mp" for a "message pointer" throughout the system's source code and documentation, an important abbreviation that needs to be captured. PEEL uses a three-step procedure to process terms similar to the process of extracting components in Basili's software factory [Caldiera and Basili 1991]. The first step is fully automated. Terms are extracted from the name of the function and any strings or comments within or immediately preceding the definition. A common stop list is applied to the extracted terms to remove common words with little semantic meaning such as "and," "of," "the," and others [Fox 1992]. PEEL users can also augment this stop list or add terms to a nonsense word list through the operations "Put this term on the Stop Word list" and "Put this term on the

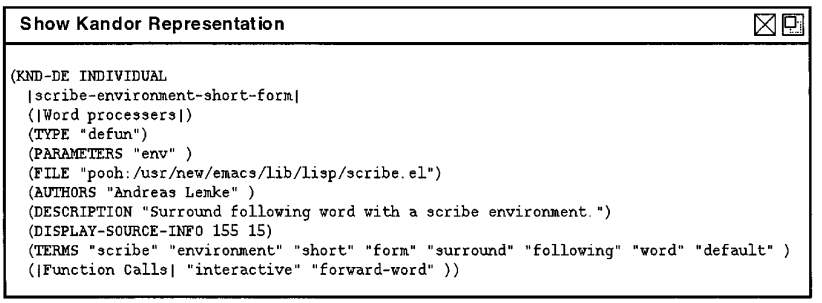

Fig. 3. The Kandor representation of an object translated by PEEL.

Nonsense Word list" in the window shown in the lower part of Figure 2. Earlier versions of PEEL extracted terms from the variable and function names, but these proved to not be very descriptive. Other researchers have had the same conclusion.[2]

The second step is to display the terms to the user. Users can remove the terms, truncate the list (a feature that proved to be useful for many components in which only the name was descriptive), and highlight the term in the source code window to show how and where the term has been used (see the pop-up window in the lower part of Figure 2). The third step allows users to add their own terms, which can include phrases (words separated by spaces) and any other punctuation.

Figure 3 shows the Kandor representation of the component named "scribe-environment-short-form" (the source code is shown in Figure 2). Much of the information is automatically extracted, although some of the information is entered once for the source code file being parsed. These include the file name and the frame to which the component belongs ("Word processors" is the frame used for all components extracted from the file "scribe.el," including "scribe-environment-short-form"). The rest of the slots are filled by extracting information from the E-Lisp source component. For example, the E-Lisp syntax allows a string after the parameters that is used by many programmers as a description of the function. This is extracted and placed in the DESCRIPTION slot, where it can be edited by PEEL users.

CodeFinder also creates hyperlinks to all functions called by a component (see the "Function Calls" attribute in Figure 3). These functions are displayed in mouse-sensitive text in the CodeFinder Interface (see the *Example of the Retrieved Items* pane of Figure 4). Displaying the immediate subsystem of a component is therefore a mouse click away.

---

[2]Personal conversation with Tom Landauer and Lynn Streeter of Bellcore. One reason for this phenomena may be that variables are closely tied to the implementation of a function and lose sight of what the function does. They are more descriptive of *how* the function works instead of *what* the function does. Since a designer's primary interest is in what functions exist to help perform a task, variable names will generally be less useful descriptors than function names and comments.

At this time, PEEL makes no attempt to assess the reusability of components. Software quality considerations dictate that we may wish to extract only components that score well on various measures such as Halstead and McCabe indicators [Caldiera and Basili 1991] or others [Poulin and Caruso 1993]. These measures have the advantage of being automatic, but suffer from assumptions based on superficial information, such as the number of operators and variables in a component. The work described here defines an infrastructure that can accommodate such metrics while providing tools to find components with specific functionality.

5. COMPENSATING FOR INCOMPLETE AND INCONSISTENT INDEXING

There are two major issues that information retrieval interfaces need to address. The first is the nature of design problems, in which people begin the process with an ill-defined notion of what is needed and how they should go about solving the problem [Belkin et al. 1982a; 1982b; Borgman 1989; Henninger 1994]. The second problem is that document indexing is often inconsistent and incomplete. Studies have shown that human indexers will disagree on terms used to index documents and that the same indexer will use different terms to index the same document at different times [Salton and McGill 1983]. Other studies have shown that people use a surprisingly diverse set of descriptors to describe the same objects [Furnas et al. 1987].

Therefore, retrieval systems that are built on the assumption that the quality of the indexing is the paramount factor in the efficacy of an information retrieval service will fall short of supporting its users. The search process is more than finding an exact match [Biggerstaff and Richter 1987; Henninger 1994]. It includes finding components that provide a partial or analogous [Maiden and Sutcliffe 1992] solution to the problem. Inconsistent indexing will always exist, making it necessary to use retrieval methods and algorithms that go beyond simplistic keyword matching schemes [Belkin and Croft 1987]. The downfall of the traditional matching strategy is that queries are viewed as precise specifications of user needs and document representations as precise descriptions of repository objects. An alternative, more realistic strategy is to assume a degree of uncertainty in document and query representations [Mozer 1984].

CodeFinder (see Figure 4) has been designed to address these issues through the employment of intelligent retrieval methods and support for query construction. "Intelligent" retrieval is provided by an associative spreading activation retrieval algorithm [Mozer 1984] that extends the exact-match paradigm to retrieve items that are associated with a query. Query construction is supported through retrieval by reformulation [Williams 1984], a technique that allows users to incrementally construct queries as they explore the information space. Empirical studies indicate that this combination of techniques achieves good performance in the face of indexing problems and ill-defined information needs [Henninger 1994]

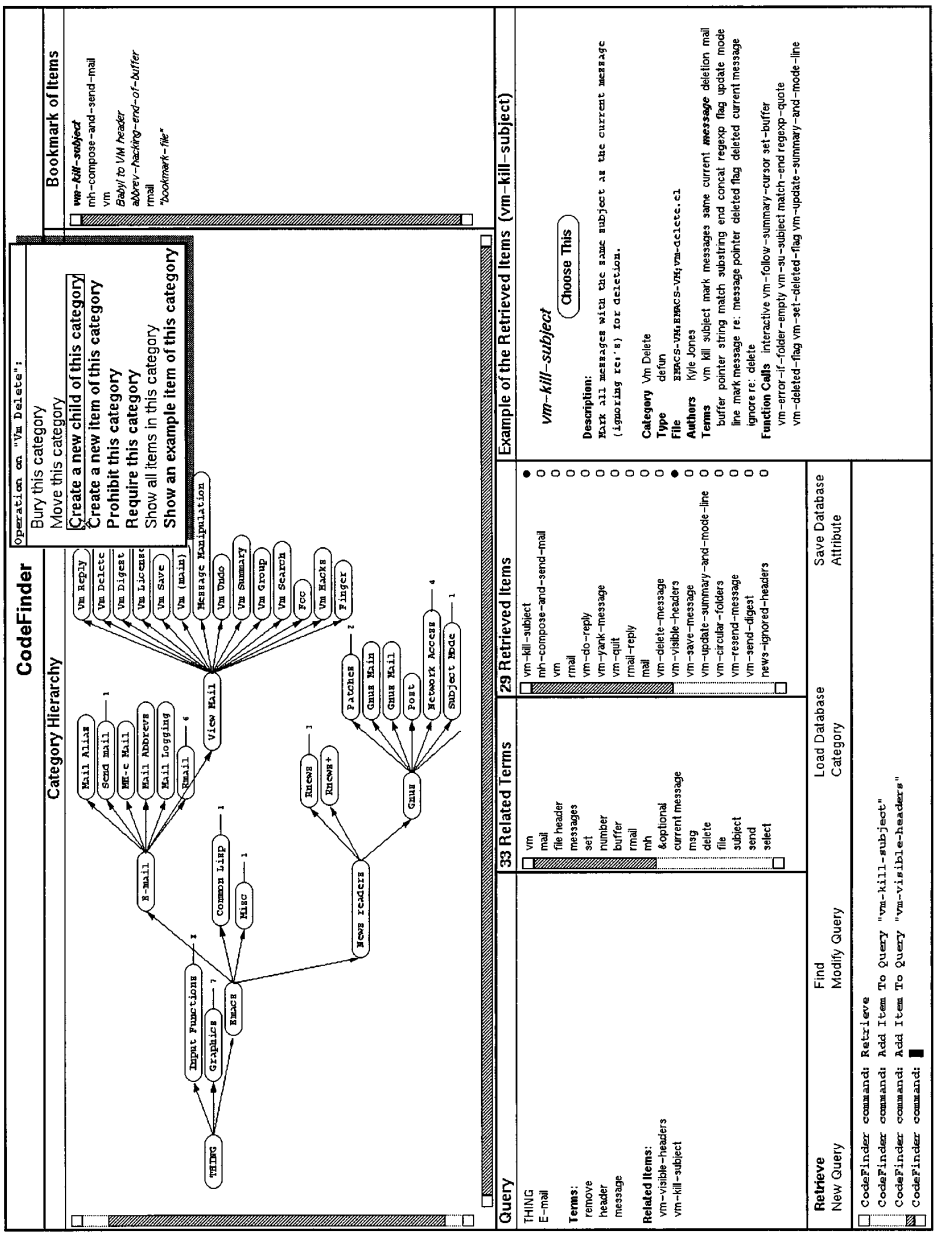

Fig. 4. CodeFinder user interface. In the interface, the *Category Hierarchy* window displays a graphical hierarchy of a repository. The *Query* pane shows the current query. The top part of the query specifies two categories ("THING" and "E-mail") to search within. The bottom part specifies key terms and related components. The *Retrieved Items* pane shows all components matching the current query, by order of relevance to the query. The *Example of the Retrieved Items* pane shows the full entry for the top matching component or a component chosen by the user. The *Bookmarks* pane holds a history view of viewed objects. The *Related Terms* pane shows terms retrieved by the query.

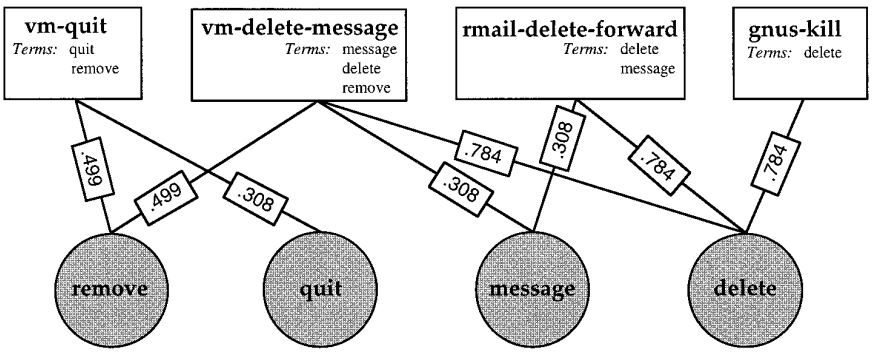

Fig. 5. An associative network. In CodeFinder the network consists of two layers of nodes: terms (represented by circles) and components (rectangles). Boxed lines are connections between components and terms, as defined by the *Terms* field of the component. Link weights are shown in link boxes.

and can work effectively with the minimally structured repository created by PEEL.

5.1 Retrieval with Spreading Activation

The repository structure created by PEEL is a collection of component representations in a frame-based organization of attribute/value pairs (see Figure 3). Given this structure, components can be retrieved by matching query terms against the TERMS fields in the repository to retrieve components (CodeFinder also uses the PARAMETERS and DESCRIPTION fields). CodeFinder uses this representation to construct an associative network (see Figure 5). An associative spreading activation process, which is based on a connectionist relaxation procedure for retrieval [Mozer 1984], is then applied to the network to retrieve information. Users specify a query consisting of either or both term and component nodes. The query nodes are given an activation value of 1.0, which remains set to that value during the spreading activation process. If a node has a positive activation value, the value is passed through each of its links; otherwise no activation is passed. Each node computes the sum of incoming activation values, modified by link weight. The sum of received activation values is then modulated by fan-in, decay, and other parameters. The resulting value is fed into a squashing function to keep activation values between boundaries of -0.2 and 1 (the asymmetrical values are needed to encourage the flow of positive activation [McClelland and Rumelhart 1981]).

For example, a query specifying the term *remove* in Figure 5 will spread a value of 0.499 (modified by various parameters described below) to the component nodes "vm-quit" and "vm-delete-message." On the next cycle, "vm-quit" and "vm-delete-message" will propagate their computed activation values to all nodes they are connected to. For example, "vm-delete-message" will activate *message* and *delete*. On the third cycle, all of the previously activated nodes will continue to propagate their values. Note

Table I. Associative Spreading Activation Parameters

Symbol	Definition
n_D	number of documents in the repository
n_T	number of distinct terms in the repository
w_{ij}	strength of connection between node i and node j
IDF	inverse document frequency
Θ_D	document decay rate
Θ_T	term decay rate
M	maximum activity level (1.0)
m	minimum activity level (−0.2)
Dt_i	number of terms in document i
df_j	number of documents indexed with term j
$\overline{Dt}$	average number of terms per document in collection
$\overline{df}$	average number of documents per term in collection
α_D	document fan-in
α_T	term fan-in

that *message* and *delete* will work together to activate "mail-delete-forward" and "gnus-kill," retrieving these components even though they are not directly indexed by terms used in the query. This is often referred to as "query expansion" [Salton and Buckley 1988]. Note also that *delete* has been identified as a kind of synonym of *remove*. Keywords are dynamically related through the items they index, compensating for inconsistent indexing through cooccurrence relationships.

The system can be allowed to cycle through the above procedure until stabilization is reached or until a maximum number of cycles are reached. Stabilization occurs when a cycle results in small changes to node activation values. With spreading activation, as in all non-Boolean retrieval systems, an unlimited number of documents may be "relevant" to some degree. But activity values quickly dissipate as activation is passed further away from query nodes. Because of computational constraints, CodeFinder allows users to choose the maximum number of cycles [Henninger 1993]. Four or five cycles will usually allow partial stabilization to occur, while still retrieving associations that are a number of items removed from the initial query.

5.1.1 *Calculating Activation Values.* Formally, spreading activation is defined by the activity levels for documents[3] and terms for discrete time units corresponding to cycles. Given the parameter definitions found in Table I, the activity level of a document D_i at time $t + 1$ is given by

$$D_i(t+1) = \begin{cases} \delta_{D_i}(t) + \Psi_{D_i}(t)(M - D_i(t)) & if\ \Psi_{D_i}(t) > 0 \\ \delta_{D_i}(t) + \Psi_{D_i}(t)(D_i(t) - m) & if\ \Psi_{D_i}(t) \leq 0, \end{cases}$$

[3]To remain consistent with standard information retrieval technology, "document" will be used in place of "component" in this section.

where $\delta_{D_i}(t)$ is the previous activation valued reduced by the document decay rate

$$\delta_{D_i}(t) = (1 - \Theta_D) D_i(t),$$

and $\Psi_{D_i}(t)$ is the net input to document i at time t, given by

$$\Psi_{D_i}(t) = \left(\frac{\overline{Dt}}{Dt_i} \right)^{\alpha_D} \sum_{j=1}^{n_T} w_{ji} U(T_j(t)),$$

where $U(x)$ is the zero-threshold identity function that does not allow negative activation values to propagate:

$$U(x) = \begin{cases} 0 & if\ x \leq 0 \\ x & if\ x > 0. \end{cases}$$

The activity level of a term unit T_j is found in the same manner, substituting term for document parameters:

$$T_j(t+1) = \begin{cases} \delta_{T_j}(t) + \Psi_{T_j}(t)(M - T_j(t)) & if\ \Psi_{T_j}(t) > 0 \\ \delta_{T_j}(t) + \Psi_{T_j}(t)(T_j(t) - m) & if\ \Psi_{T_j}(t) \leq 0, \end{cases}$$

where $\delta_{T_j}(t)$ is the previous activation valued reduced by the document decay rate,

$$\delta_{T_j}(t) = (1 - \Theta_T) T_j(t),$$

and $\Psi_{T_j}(t)$ is the net input to term j at time t, given by

$$\Psi_{T_j}(t) = \left(\frac{\overline{df}}{df_j} \right)^{\alpha_T} \sum_{i=1}^{n_D} w_{ij} U(D_i(t)).$$

5.1.2 *Parameters of the Spreading-Activation Algorithm.* The activity level of a node is the combination of the previous activation reduced by a decay factor and the net input scaled by the difference between the previous activation value and the maximum or minimum. Retaining part of the previous activity reduces wild fluctuations that occur when activity reaches a node at different times, essentially integrating evidence for a node across all cycles in the process [Belew 1987]. This value is then fed into an *arctan*() function to keep values between the maximum and minimum. The decay factor Θ causes a node to lose a fixed percentage of activation each cycle, resulting in an exponential decay of activation value over time. Nodes that receive some initial activation, but lack confirming evidence in the form of positive feedback from activated nodes, will decay over time toward a resting value of 0. This prevents an "inertia" phenome-

non in which nodes receiving activation retain it forever. Because terms will normally outnumber documents, and because terms serve as multiple sources to find relevant documents, decay rates on terms need to be set higher than documents. Decay rates of 0.25 for terms (Θ_T) and 0.1 for documents (Θ_D) have been shown experimentally to work well [Henninger 1993; Mozer 1984].

The net input, Ψ, is modified by a fan-in factor, x^{α}, to prevent nodes with large numbers of connections from receiving unduly large amounts of activation. Dividing the average number of terms per document by the number of terms for a given node inflates the net input for nodes with less terms than average and reduces net input for those with more. Raising this factor by a fan-in exponent, α, magnifies the effect and provides a parameter that can be adjusted until the bias for nodes with large numbers of connections is eliminated. Experiments with CodeFinder's repository found that $\alpha_D = 0.3$ and $\alpha_T = 0.2$ effectively removed fan-in bias [Henninger 1993].

5.1.3 *Link Weights.* A critical part of the structure in spreading-activation systems is the link weighting between nodes in the associative network. This controls the amount of information passed during spreading-activation cycles. Previous spreading-activation systems have defined a constant link weight between nodes [Mozer 1984]. This is based on the false conjecture that all terms are equally related to an item. For example, if the terms "reply" and "mail" are used to index the item *vm-reply* it would be safe to say that "reply" is more descriptive of the function. "Reply" is a better *discriminator*—it better describes the item and should retrieve *vm-reply* with a higher probability than "mail." This kind of discrimination can be accomplished in an associative net by varying the weight in proportion to the discrimination value of a term, with better discriminators having higher weights that will result in the spread of larger activation values.

Term-to-item discrimination is difficult to define in a generalized manner and will be difficult to apply to every term-component relationship in a large repository. Fortunately, the empirical observation that there is a tendency for less frequent terms to be more precise (in terms of recall and precision) has led to effective approximations [Furnas et al. 1987; Sparck-Jones 1972]. If a term has a low frequency (occurs seldom in the information space) then it is a good discriminator of the items it indexes. For example, if an item is indexed by the terms A and B, and A indexes 100 other items, but B indexes only 2 others, then B is quite unique to the item and should retrieve it with a rather high probability. This method, called *inverse document frequency* (IDF), has proven quite effective in retrieval systems for a number of domains [Belkin and Croft 1992; Salton and Buckley 1988].

In CodeFinder, the weight of connection from the document to descriptors

w_{ji} is given by (weights are symmetrical)

$$w_{ji} = w_{ij} = \begin{cases} 0 & \text{if no connection between } i \text{ and } j \\ \text{IDF} = \dfrac{\log(n_D/df_j)}{\log(n_D)} & \text{if } i \text{ and } j \text{ connected.} \end{cases}$$

This is a modified form of the standard IDF weighting scheme, which is $\log(n_D/df_j)$ [Sparck-Jones 1973]. The modification, dividing the standard IDF by the log of the total number of documents, normalizes weight values to be between 0 and 1.0. Note that the weight of a term that indexes one document is 1.0 (i.e., $\log(n_D)/\log(n_D)$).

This scheme gives one weight to each term in the repository. A better discrimination measure would be to recognize that some terms are better discriminators for certain items and should not have a uniform weight across all items it indexes. One measure for this is *term frequency* (TF), which is the frequency of the term in a given item or document [Salton and Buckley 1988]. CodeFinder uses only the IDF measure because software source code objects do not have the linguistic regularity of large text documents that TF measures rely on. CodeFinder achieves individual term-to-document weighting through relevance feedback mechanisms (see Section 6.3).

5.1.4 *Defining Relationships through Content-Induced Structure.* The power of the combination of PEEL and CodeFinder is that indexing does not need to be exhaustive for CodeFinder to effectively locate software objects. A minimal indexing with PEEL combined with the spreading activation and browsing tools of CodeFinder adequately supports the process of locating relevant source code, which has been empirically validated in user studies [Fischer and Nieper-Lemke 1989; Foltz and Kintsch 1988; Henninger 1994]. The associative spreading-activation model uses common connections between keywords and components to "induce" relationships between components and terms [Mozer 1984], compensating for queries using improper terminology.

It is the structure of the repository, and not predefined term or component relationships, that spreading activation uses to retrieve information and induce relationships between components. A particularly salient way to demonstrate this effect is to show the related keywords retrieved by single-word queries. Figure 6(a) shows that the query *delete* will retrieve a number of synonyms. The semantic relationship between *delete* and the components retrieved has not been programmed into the system. Given only the structure of the repository in which the terms *delete*, *remove*, and *kill* cooccur between components, spreading activation is able to induce the semantic relationship between these words. This is referred to as "content-induced structure" [Halasz 1988]. Figure 6(c) shows that when the term *open* is specified, terms related to things that have open operations, such as

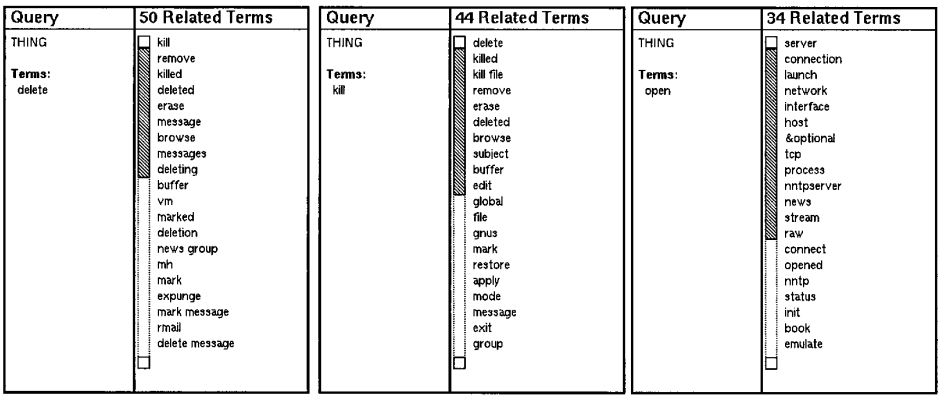

Fig. 6. Capturing word relationships through content-induced structure. The highest-rated terms (most relevant to the query) in the *Related Terms* pane, appearing in order from highest (top) to lowest (bottom) activation value, are semantically related to the queries; (a) shows the results of the query with only *delete* specified; (b) uses the term *kill*; and (c) uses *open*.

server, *connection*, *host*, and others, are retrieved in addition to synonyms such as *launch* and *opened* (note that these terms would not normally be considered synonyms, but are terms that frequently cooccur in programming idioms). The screen images shown in Figure 6 were retrieved directly from the Emacs Lisp repository created by PEEL, without any augmentation of the repository, and are only a sampling of the kinds of semantic relationships that can be derived by spreading activation. To the extent that adequate associative structure exists, these kinds of relationships will be found.

Not only is spreading activation able to find semantically related words, it captures idiosyncratic term usage in the repository. For example, in Figure 6(a), the relationship between *delete* and *kill* captures a programming idiom in which the term *kill* often refers to deleting an object. In addition, one of the programs, View Mail (by Kyle Jones), uses the term *delete* when deleting messages and *kill* when referring to removing files (Figure 6(b)). Therefore, as shown in the figure, the *delete* query retrieved terms related to messages while the *kill* query found a number of words related to files. This kind of idiosyncratic term usage will be missed by traditional thesaurus methods based on equivalence classes for words, but are important to domains that generate idiomatic terminology to describe complex concepts, such as software development, medical fields, and others. Associative spreading activation uses the structure of the repository to find idiomatic word relationships with no additional labor-intensive effort to predefine word relationships. These computed word relationships can then be displayed in the *Related Keywords* pane (see Figure 4) to provide cues for reformulating the query.

5.2 Retrieval by Reformulation

Beyond soft-matching algorithms, retrieval systems can reduce the effects of a poor repository structure by supporting the process of query construction. Retrieval by reformulation is a method that supports incremental query formation by building on query results [Williams 1984]. Each time a user specifies a query, the system responds with query reformulation cues that give users an indication of how the repository is structured and what terms are used to index objects. Users can then incrementally improve a query by critiquing the results of previous queries. This supports the refinement of information needs by accumulating knowledge acquired during the search process while narrowing in on a solution. Rabbit [Williams 1984] and Helgon [Fischer and Nieper-Lemke 1989] are examples of retrieval systems based on the retrieval by reformulation paradigm.

CodeFinder supports retrieval by reformulation by providing a number of retrieval cues in its interface. The most important of these is the example components appearing in the *Example of the Retrieved Items* pane (see Figure 4). This pane shows the full representation of a component, allowing the user to choose any of the attributes for inclusion or exclusion in a query. Retrieval by reformulation is accomplished through mouse action that allows users to simply point and click on components to be included in the query. For example, if a user wanted to add the term "subject" shown in the "Terms" attribute of the component "vm-kill-subject" in Figure 4 they would simply click-left on the term. The term would be placed in the Terms field of the Query pane, and the "Retrieve" button in the command pane would be displayed in bold, indicating that the query had changed. Choosing the "Retrieve" button would then retrieve information based on the new query.[4]

CodeFinder enhances the retrieval-by-reformulation paradigm in three ways. First, items are automatically placed in the appropriate part of the query. As opposed to query languages, where users are responsible for both the structure and content of queries, users are spared the cognitive overhead of deciding where to put an attribute, category, or term. Second, CodeFinder imposes a ranking criteria on the retrieval set, automatically displaying the highest-rated item for the query. This is important because the example provides cues that characterize what kinds of items are retrieved by the query and how it can be improved. The better the example, the easier it is to converge on a satisfactory solution. Selecting a good example has not been addressed by previous retrieval by reformulation systems. Helgon, for example, organized the retrieval set by alphabetic order and displayed the first example in this ordering [Fischer and Nieper-Lemke 1989]. Third, CodeFinder displays terms in the *Related Terms* pane

[4]The two-step process was used because retrieval will take a nontrivial amount of time in large repositories. If users wish to make multiple changes to a query, they would have to wait each time a change is made. The "Retrieve" button puts users in control of when this potentially time-consuming operation takes place.

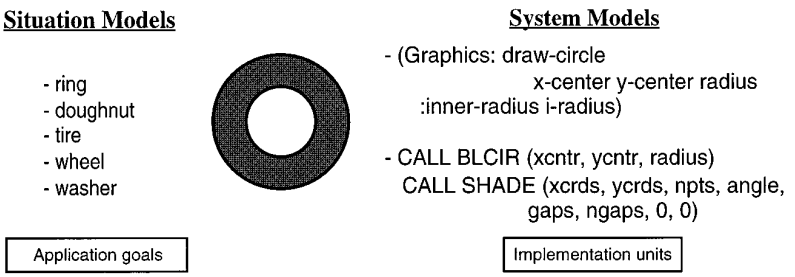

Fig. 7. Situation and system models.

(shown in Figures 4 and 6). These terms are also ranked by order of relevance to the query and can be used to refine the query.

6. SUPPORTING THE INCREMENTAL REFINEMENT OF COMPONENT REPOSITORIES

Once components have been captured with minimal structure, it is necessary to evolve the structure to meet the needs of a development organization. This is necessary not only to improve the indexing and structure of the repository, but also to keep information current as knowledge in the organization evolves [Henninger et al. 1995]. Even with innovative and intelligent retrieval tools, a repository's effectiveness is influenced by the quality of the indexing. The problem is that indexing "quality" is a relative measure, depending on the retrieval context arising from different information needs and perspectives that differ both for individuals with different information needs at different times, as well as for different users with different backgrounds [Harter 1992]. Therefore, effective repository structures cannot be designed up front; they must be allowed to evolve in the context of use.

The ultimate goal of a component repository is to create an "indexing scheme similar to the knowledge structures possessed by most programmers working in an application area" [Curtis 1989]. To date, little research has been directed at characterizing the nature of those knowledge structures. One potentially useful line of research has studied the relationship between how people model problem and solution spaces [Kintsch and Greeno 1985]. This theory distinguishes between the *situation model* (a designer's model of the application domain and the specific problem he or she is faced with) and the *system model* (the set of resources provided by the system, programming languages, and environments, for example). The relationship between these models, shown in Figure 7, is one of translation. The initial, high-level conceptualization by users of the problem needs to be translated into language that was used to index the desired items. The effectiveness of a repository can be improved by minimizing the mismatch between situation and system models. This can be a formidable problem, as demonstrated in Figure 7, which was derived from an empirical study that asked people for keywords to describe the given object [Henninger 1994].

Note that while users will often conceptualize the retrieval problem in terms of their *application goal*, system terminology refers to how the object is *implemented*. Users unfamiliar with the implementation model will have difficulty finding the desired functionality. It is not enough to merely have some structure in place; it has to be a structure that the user can understand and use, something that supports the user's model of the system.

Curtis notes that empirical studies have shown that as programmers gain experience, their understanding of the knowledge in a particular domain gravitates toward a common structure [Curtis 1989]. It has also been empirically shown that indexing structures in adaptive systems converge toward common themes in groups with similar interests [Belew 1987]. While it is difficult to anticipate all design needs or uses for design artifacts in advance, a group of users working in a moderately homogeneous environment will tend to approach problems using similar terminology. Studies have shown that a great deal of effort in a project is directed at creating a mutual understanding of the problem among the developers [Walz et al. 1993]. The best opportunity to capture typical use situations is to provide the means to easily modify the repository during use. This provides critical insight into how designers view their domain—the characteristics of their situation models.

6.1 Adapting the Repository in the Context of Use

Adaptive techniques to improve the repository are applied by recording user actions and incrementally changing representations based on those actions. As users interact with the system, the system will migrate toward an indexing scheme using the kinds of application goals typically encountered by users. The overall goal is to have the system gradually build a consensual representation of its user's knowledge [Rose and Belew 1991, p. 3] that can change as the knowledge in an organization evolves.

A Bellcore study of people choosing terms to describe common objects, cooking recipes, editor commands, and other items revealed that the probability of two people choosing the same keyword for these objects is between 10 and 20% [Furnas et al. 1987]. Using 15 aliases will achieve 60–80% agreement, and 30 aliases can get up to 90% agreement. This led to the notion of "unlimited aliasing," in which any term applied by a user to describe an object should be used as a descriptor (index term) for that object [Furnas et al. 1987]. Note that aliasing is different than traditional thesaurus methods using equivalence classes for terms, because in aliasing the terms point directly to the objects, not to other terms in the thesaurus.

One of the problems with an unlimited aliasing approach is the potential of precision problems. If terms are used more often to describe objects in the repository, then the probability of a system retrieving unwanted objects increases. Studies of retrieval systems that use large numbers of aliases have shown that precision problems are less severe than theory would expect [Gomez et al. 1990]. This occurs because in practice more terms per

object do not necessarily imply more objects per term, and the theoretical notion of precision (the ratio of irrelevant objects retrieved to the total retrieval set) matters less than recall (the ratio of relevant objects retrieved to the total retrieval set) in an interactive retrieval environment. In other words, as long as relevant objects are retrieved, and people are able to find and peruse these objects, it matters less how many irrelevant objects are retrieved.

One method to achieve unlimited aliasing without unwanted precision problems is adaptive indexing [Furnas 1985]. This method collects term usage data interactively, in the process of real information-seeking sessions, by adding terms to an object representation when they are used in a search that was satisfied by the document. CodeFinder supports unlimited aliasing and adaptive indexing by tracking term usage during a retrieval session and allowing users to associate those terms with the indexing representation of a selected item. During the process of constructing a query and browsing the information space, a user might remove a number of terms from the query and enter terms that were not in the system lexicon (in which case an error message warning of this deficiency is posted). These terms can be used as evidence of a situation model, an initial way of thinking about the retrieval problem. There may also be some terms in the current query that are not in the item's current representation. Adding all of these terms to a chosen item may enhance the future retrievability of the item, but we must guard against situations in which misspellings or undesired terms are used in the retrieval session.

CodeFinder addresses this issue by displaying the information to the user for inspection and possible modification. These terms are presented to the user when they click on the "Choose This" button in the "Example of the Retrieved Items" pane (see the *Example of the Matching Items* pane in Figure 4). Clicking on this button can have a number of semantics, such as placing the example in an editor buffer or invoking other CASE tools to analyze the component [Fischer et al. 1991]. When this occurs, an adaptive indexing session is invoked, as shown in Figure 8. Users are given the option to add these items to the current representation of the chosen example and are allowed to enter other terms as needed.

Repository users can also modify the repository structure used by the Kandor-based part of CodeFinder through pull-down menus such as the one shown in the middle-top of Figure 3 [Henninger 1995]. This kind of modification is perhaps best handled by a repository librarian, but is an important facility that enables evolution of the repository.

6.2 Adjusting Link Weights with Relevance Feedback

In addition to assigning new terms to a component representation, CodeFinder uses relevance feedback to adjust the term-component link weight. Relevance feedback is typically used to reformulate a query, usually by adding a document's representation to the query [Salton and Buckley 1990]. CodeFinder accomplishes this by allowing users to click on the

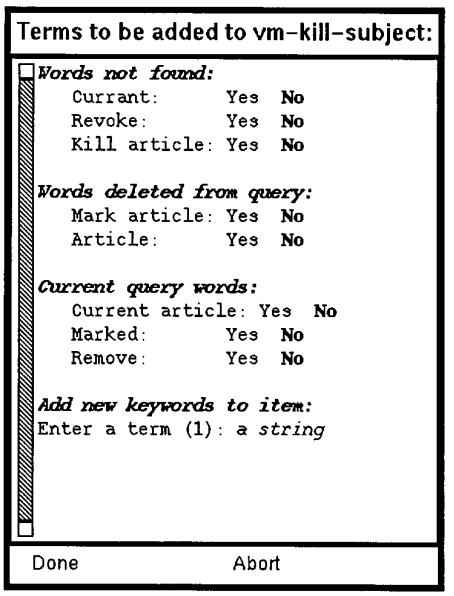

Fig. 8. Adaptive indexing in CodeFinder.

circles on the right-hand side of the Retrieved Items pane (see Figure 4). Each item chosen is placed in the query (see the Related Items portion of the Query pane), where it is used as part of the spreading-activation query.

CodeFinder also uses this action as an indication that the chosen components are relevant to the terms shown in the Terms portion of the Query pane. Strengthening the link weight between these components and terms will increase the probability that similar queries in the future will get better retrieval results. A *localized reinforcement* [Belew 1987] learning technique, which normally adjusts weights so that activating some input nodes results in a target vector on the output, is used to strengthen weights. User feedback is used in CodeFinder in lieu of a target. The association we wish to strengthen is between terms in the query and components chosen in relevance feedback. Therefore, if a component chosen in relevance feedback is indexed by any of the terms in the query, then each of the weights are adjusted with the following learning rule [Rose and Belew 1991, p. 21]:

$$w'_{ij} = w_{ij} + \eta f_i a_j,$$

where a_j is the activation value of the document node j; f_i is the feedback given to node i by the user (a binary yes/no value in CodeFinder); and η is the learning rate, usually a value between 0 and 1.0. The values are adjusted by an *arctan*() squashing function to keep values between -1.0 and 1.0. Since "the knowledge is in the link weights" [Smolensky 1988], this is potentially an important contribution to the effectiveness of spreading activation. Unfortunately, evaluating the effectiveness of this method runs

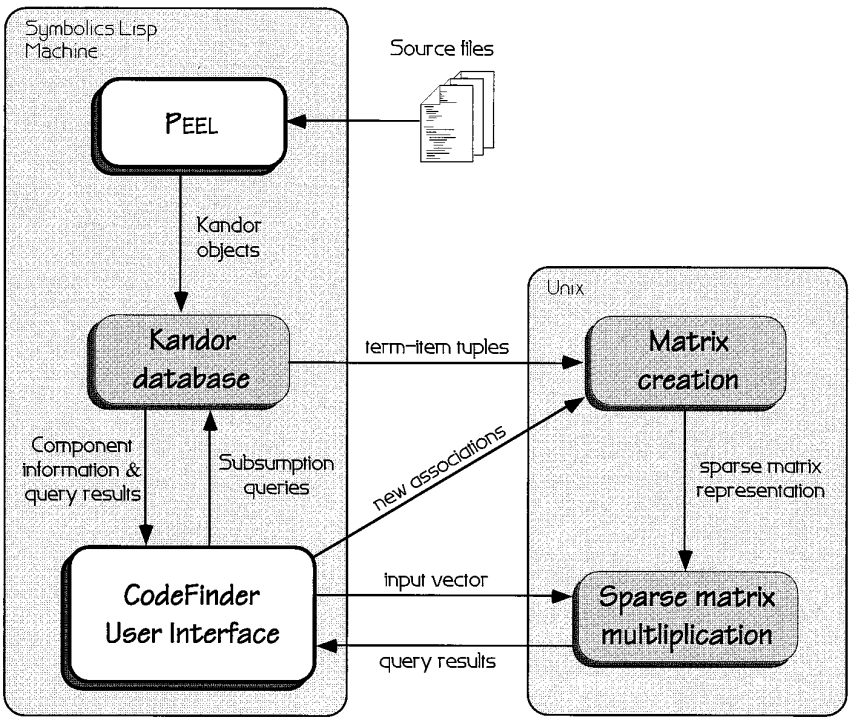

Fig. 9. CodeFinder architecture.

into the "credit assignment" problem of which weights were responsible for the results and which should be changed to improve performance. However, empirical studies have shown that retrieval performance does indeed improve over time, especially when used in groups with homogeneous interests [Belew 1987].

7. DESIGN AND EVALUATION OF CODEFINDER

7.1 Implementation

The general architecture of CodeFinder is shown in Figure 9. CodeFinder's interface was built on a Symbolics Lisp Machine. For reasons of efficiency and scaling the repository, spreading-activation calculations are implemented as C routines executed on a Unix machine. A socket interface is used to pass information between the different operating systems. PEEL takes E-lisp source files as inputs and produces Kandor objects. To create a matrix, CodeFinder extracts terms from Kandor objects and creates a series of tuples of the form {term, item}, one for each term in each database object. On the Unix side, each tuple is inserted into a sparse matrix representation, and statistics are accumulated for the spreading-activation parameters.

The implementation of spreading activation is straightforward matrix multiplication with adjustments for algorithm parameters. Connections between terms and documents are represented in a $n \times n$ connection matrix where n is the vector containing all documents and terms. The initial query is represented in a vector of length n with query nodes (terms or documents) set to 1.0. This vector is multiplied by the matrix, resulting in a vector that sums the net input to all nodes connected to the query. The nodes are then adjusted by algorithm parameters. The resulting vector is multiplied by the connection matrix to compute the next cycle. Matrix multiplication, especially with large matrices, is an expensive operation. Fortunately, associative networks sparsely populate the matrix, allowing greater computational efficiency with sparse matrix multiplication algorithms. This is especially true of the structured associative networks used in CodeFinder, which do not allow links between terms and between components. Response time of less than one second is achieved with the Emacs Lisp repository using a DECstation 5000 and a rudimentary sparse matrix multiplier. The computational complexity of the algorithm is proportional to the sparse matrix multiplication algorithm that is used to implement spreading activation. Faster hardware and better sparse matrix multiplication algorithms could easily accommodate repositories with tens of thousands of items with acceptable performance.

CodeFinder also allows for the storage of the matrix representation on the Unix side so that the matrix creation step can be omitted. Updates to the matrix through adaptive indexing facilities (see Section 6) are handled by sending the tuples to the matrix creation module which inserts the tuple into the matrix and updates the parameters. If the new term is unique, i.e., does not appear in the repository, nothing else is done. But if an existing term is added to a component, the new term-item association changes the link weight for that term (keep in mind that weights are determined by the IDF measure). The weight must be adjusted for each component with which the term is associated.

7.2 Empirical Evaluation of CodeFinder

An experiment was conducted to assess the effectiveness of CodeFinder's interface and retrieval methods with minimal structures (i.e., structure derived from PEEL without the benefit of adaptive indexing) [Henninger 1993; 1994]. Subjects were given tasks that involved finding software components and functionality. Tasks were adapted from observations of people developing software in the Emacs Lisp environment. CodeFinder was compared against two other software component retrieval systems that used direct-matching retrieval algorithms and had limited capabilities for reformulating queries. The systems were assessed on their ability to help subjects solve problems along dimensions of problem definition (well defined, directed, and ill defined) and vocabulary match (match, mismatch).

The results showed that differences between the systems were more pronounced when the tasks were in the ill-defined or vocabulary mismatch

categories [Henninger 1994]. CodeFinder scored best in these categories, with subjects finding more items faster. The results indicate that subjects had problems with systems using straightforward matching algorithms when problems were ill defined or when the vocabulary mismatched. The greater degree of success with CodeFinder indicates that soft-matching retrieval algorithms, such as spreading activation, coupled with query construction techniques are necessary to adequately support the kinds of tasks typically encountered by developers searching for information about reusable components.

It is important to underscore the fact that these results are a consequence of CodeFinder's overall interface, not just the spreading-activation algorithm. The experiment was designed to evaluate the interface, not retrieval algorithms, and therefore the evaluation criteria differed from traditional recall and precision measures [Salton and McGill 1983], focusing instead on a subject's ability to use the system's features to find relevant information. Our results thus agree with other studies concluding that retrieval methods alone are insufficient to gain performance advantage [Frakes and Pole 1994]. Because we were able to control for effects from spreading activation and retrieval by reformulation [Henninger 1994], our hypothesis is that CodeFinder's better performance was due at least in part to the coupling of these methods. More studies are needed to verify these results and improve our understanding of our observations.

8. CONCLUSIONS AND FUTURE DIRECTIONS

The work described in this article is rooted in the need for software development tools that support the process of finding components for reuse. Although it is true that perfectly constructed and indexed component repositories make relevant components easy to find, the reality is that pragmatic and inherent circumstances prevent the creation of such repositories. The ill-defined nature of design problems is one such problem that makes it difficult to index components for future use. Another problem is that the unfamiliar and esoteric vocabulary often used in the software design process makes it difficult for potential reusers to find useful components.

It therefore is both difficult and costly to anticipate all the needs for which a given component will be used. Even if perfection could be achieved, the fact that people use diverse vocabulary to describe diverse information needs will prevent users from always finding the proper components. In addition to being costly, the structures required for retrieval are often static and unable to adapt to dynamic development contexts. Once the structures are created, they are "set in stone," changeable only through complex procedures which often involve redesigning the entire repository. Methods are needed that (1) provide adequate retrieval effectiveness with minimal indexing and structuring efforts, allowing organizations to take advantage of valuable assets accumulated from previous development efforts without a large up-front investment in reusability, and (2) help the

repository evolve with the changing needs of people in a software development organization.

This article presents an approach to supporting a kind of "lifecycle" for repositories. Repositories are initially seeded with structure and index terms through PEEL, a tool that extracts repository information from Emacs Lisp Source code. CodeFinder, which supports the retrieval process through retrieval by reformulation and spreading activation, is then used to help users find components in the face of less-than-perfect repository structures and indexing. While using CodeFinder to find components, users are given the opportunity to add structure and index terms to the repository, improving the repository to reflect the kinds of problems repository users typically encounter. This process has the advantage of avoiding costly repository-populating efforts while capturing information that may be useful to people encountering similar information needs in the future.

Empirical studies have shown that CodeFinder is able to adequately support the process of finding relevant software components, even with the minimal structures and indexing performed by PEEL [Henninger 1993; 1994]. The results of this study merely provide some evidence for the effectiveness of the repository tools provided by CodeFinder. More studies are needed to better understand the nature of the retrieval process in the context of software development. Longitudinal studies in the context of software development organizations are also needed to assess the utility and limitations of the adaptive facilities in CodeFinder. Studies of this nature promise to bring a better understanding of how to satisfy information needs in problem-solving contexts and how repositories can evolve to become more effective over time.

The studies have also pointed out that a number of improvements are needed in the PEEL/CodeFinder tool set before it can become a viable tool for development organizations. At this time no attempt has been made to assess the reusability of components or capture other component metrics. Work is under way to apply the methods described in this article to larger-gain software components and nonsource code artifacts [Henninger 1997; Henninger et al. 1995]. More studies are also needed to compare CodeFinder to other repository retrieval methods such as faceted classification [Prieto-Díaz and Freeman 1987], although the up-front repository construction costs of these methods make such comparisons difficult.

In order for software reuse methodologies to be feasible and effective, methods are needed to exploit valuable assets that exist in development organizations in the form of existing code modules that have been developed and saved with no thought of reuse. Tools for building a repository must be developed that extract these modules and represent them in a repository form that is amenable to a specific set of retrieval tools. The work presented here takes some steps toward addressing these issues. Future work is needed to refine these ideas and subject them to rigorous empirical studies to assess their effectiveness and learn more about the nature of component repositories built to support software reuse techniques.

ACKNOWLEDGMENTS

I thank members of Gerhard Fischer's Human-Computer Communication group at the University of Colorado, especially Peter Foltz, Gerhard Fischer, Walter Kintsch, David Redmiles, and Curt Stevens, with whom many discussions were shared about the ideas discussed in this article. Generous donations of reusable software by Helga Nieper-Lemke and Jonathan Bein gave CodeFinder its initial start.

REFERENCES

BARNES, B. H. AND BOLLINGER, T. B. 1991. Making reuse cost-effective. *IEEE Softw. 8*, 1, 13–24.

BATORY, D., SINGHAL, V., THOMAS, J., DASARI, S., GERACI, B., AND SIRKIN, M. 1994. The GenVoca model of software-system generators. *IEEE Softw. 11*, 5, 89–94.

BELEW, R. K. 1987. Adaptive information retrieval: Machine learning in associative networks. Ph.D. dissertation, Tech. Rep. 4, Univ. of Michigan, Ann Arbor, Mich.

BELKIN, N. J. AND CROFT, W. B. 1987. Retrieval techniques. *Ann. Rev. Inf. Sci. Tech. 22*, 109–145.

BELKIN, N. J. AND CROFT, W. B. 1992. Information filtering and information retrieval: Two sides of the Same Coin? *Commun. ACM 35*, 12 (Dec.), 29–38.

BELKIN, N. J., ODDY, R. N., AND BROOKS, H. M. 1982a. Ask for information retrieval: Part 1. *J. Doc. 38*, 2, 61–71.

BELKIN, N. J., ODDY, R. N., AND BROOKS, H. M. 1982b. Ask for information retrieval: Part 2. *J. Doc. 38*, 3, 145–163.

BIGGERSTAFF, T. J. 1992. An assessment and analysis of software reuse. *Adv. Comput. 34*, 1–57.

BIGGERSTAFF, T. J. AND RICHTER, C. 1987. Reusability framework, assessment, and directions. *IEEE Softw. 4*, 2, 41–49.

BLACKSHAW, L. AND FISCHOFF, B. 1988. Decision making in online searching. *J. Am. Soc. Inf. Sci. 39*, 6, 369–389.

BLAIR, D. C. AND MARON, M. E. 1985. An evaluation of retrieval effectiveness for a full-text document-retrieval system. *Commun. ACM 28*, 4 (Apr.), 289–299.

BOOCH, G. 1987. *Software Components with ADA*. Benjaman Cummings, Menlo Park, Calif.

BORGMAN, C. L. 1989. All users of information retrieval systems are not created equal: An exploration into individual differences. *Inf. Process. Manage. 25*, 3, 237–251.

BRACHMAN, R. J., MCGUINNESS, D. L., PATEL-SCHNEIDER, P. F., RESNICK, L. A., AND BORGIDA, A. 1991. Living with CLASSIC: When and how to use a KL-ONE-like language. In *Principles of Semantic Networks: Explorations in the Representation of Knowledge*, J. F. Sowa, Ed. Morgan Kaufmann, San Mateo, Calif., 401–456.

BURTON, B. A., ARAGON, R. W., BAILEY, S. A., KOEHLER, K. D., AND MAYES, L. A. 1987. The reusable software library. *IEEE Softw. 4*, 4, 25–33.

CALDIERA, G. AND BASILI, V. R. 1991. Identifying and qualifying reusable software components. *Computer 24*, 2, 61–70.

CAMERON, D. AND ROSENBLATT, B. 1991. *Learning GNU Emacs*. O'Reilly and Assoc., Sebastopol, Calif.

CHEN, P. S., HENNICKER, R., AND JARKE, M. 1993. On the retrieval of reusable software components. In *Advances in Software Reuse* (*Proceedings of the 2nd International Workshop on Software Reusability*). IEEE Computer Society Press, Los Alamitos, Calif., 99–108.

CURTIS, B. 1989. Cognitive issues in reusing software artifacts. In *Software Reusability*. Vol. 2, *Applications and Experience*, T. J. Biggerstaff and A. J. Perlis, Eds. Addison-Wesley, Reading, Mass., 269–287.

DEVANBU, P., BRACHMAN, R. J., SELFRIDGE, P. G., AND BALLARD, B. W. 1991. LaSSIE: A knowledge-based software information system. *Commun. ACM 34*, 5 (May), 34–49.

FISCHER, G. AND NIEPER-LEMKE, H. 1989. HELGON: Extending the retrieval by reformulation paradigm. In *Human Factors in Computing Systems, CHI '89 Conference Proceedings*. ACM, New York, 333–352.

FISCHER, G., HENNINGER, S. R., AND REDMILES, D. F. 1991. Cognitive tools for locating and comprehending software objects for reuse. In the *13th International Conference on Software Engineering*. ACM, New York, 318–328.

FISCHER, G., MCCALL, R., OSTWALD, J., REEVES, B., AND SHIPMAN, F. 1994. Seeding, evolutionary growth and reseeding: Supporting the incremental development of design environments. In *Proceeding of the Conference on Computer-Human Interaction* (*CHI '94*). ACM, New York, 292–298.

FOLTZ, P. W. AND KINTSCH, W. 1988. An empirical study of retrieval by reformulation on HELGON. Tech. Rep. 88-9, Inst. of Cognitive Science, Univ. of Colorado, Boulder, Colo.

FOX, C. 1992. Lexical analysis and stoplists. In *Information Retrieval: Data Structures and Algorithms*, W. B. Frakes and R. Baeza-Yates, Eds. Prentice-Hall, Englewood Cliffs, N.J., 102–130.

FRAKES, W. B. AND GANDEL, P. B. 1990. Representing reusable software. *Inf. Softw. Tech. 32*, 10, 653–664.

FRAKES, W. B. AND NEJMEH, B. A. 1987. Software reuse through information retrieval. In *The 20th Hawaii International Conference on System Sciences*. IEEE Computer Society Press, Los Alamitos, Calif., 530–535.

FRAKES, W. B. AND POLE, T. 1994. An empirical study of representation methods for reusable software components. *IEEE Trans. Softw. Eng. 20*, 8, 617–630.

FURNAS, G. W. 1985. Experience with an adaptive indexing scheme. In *Human Factors in Computing Systems, CHI '85 Conference Proceedings*. ACM, New York, 131–135.

FURNAS, G. W., LANDAUER, T. K., GOMEZ, L. M., AND DUMAIS, S. T. 1987. The vocabulary problem in human-system communication. *Commun. ACM 30*, 11 (Nov.), 964–971.

GOMEZ, L. M., LOCHBAUM, C. C., AND LANDAUER, T. K. 1990. All the right words: Finding what you want as a function of richness of indexing vocabulary. *J. Am. Soc. Inf. Sci. 41*, 8, 547–559.

HALASZ, F. G. 1988. Reflections on notecards: Seven issues for the next generation of hypermedia systems. *Commun. ACM 31*, 7 (July), 836–852.

HARTER, S. P. 1992. Psychological relevance and information science. *J. Am. Soc. Inf. Sci. 43*, 9, 602–615.

HENNINGER, S. 1994. Using iterative refinement to find reusable software. *IEEE Softw. 11*, 5.

HENNINGER, S. 1995. Information access tools for software reuse. *J. Syst. Softw. 30*, 3, 231–247.

HENNINGER, S., LAPPALA, K., AND RAGHAVENDRAN, A. 1995. An organizational learning approach to domain analysis. In the *17th International Conference on Software Engineering*. ACM Press, New York, 95–104.

HENNINGER, S. R. 1993. Locating relevant examples for example-based software design. Ph.D. dissertation, Univ. of Colorado, Boulder, Colo.

KINTSCH, W. AND GREENO, J. G. 1985. Understanding and solving word arithmetic problems. *Psychol. Rev. 92*, 109–129.

LAKOFF, G. 1987. *Women, Fire, and Dangerous Things: What Categories Reveal about the Mind*. The University of Chicago Press, Chicago, Ill.

MAAREK, Y. S. AND SMADJA, F. A. 1989. Full text indexing based on lexical relations, an application: Software libraries. In *Proceedings of SIGIR '89*. ACM, New York, 198–206.

MAAREK, Y. S., BERRY, D. M., AND KAISER, G. E. 1991. An information retrieval approach for automatically constructing software libraries. *IEEE Trans. Softw. Eng. 17*, 8, 800–813.

MAIDEN, N. A. AND SUTCLIFFE, A. G. 1992. Exploiting reusable specifications through analogy. *Commun. ACM 35*, 4 (Apr.), 55–64.

MCCLELLAND, J. L. AND RUMELHART, D. E. 1981. An interactive activation model of context effects in letter perception: Part 1: An account of basic findings. *Psychol. Rev. 88*, 5, 375–407.

MOZER, M. C. 1984. Inductive information retrieval using parallel distributed computation. ICS Rep. 8406, Inst. for Cognitive Science, Univ. of California—San Diego, La Jolla, Calif.

OSTERTAG, E., HENDLER, J., PRIETO-DÍAZ, R., AND BRAUN, C. 1992. Computing similarity in a reuse library system: An AI-based approach. *ACM Trans. Softw. Eng. Methodol. 1*, 3, 205–228.

PATEL-SCHNEIDER, P. F., BRACHMAN, R. J., AND LEVESQUE, H. J. 1984. ARGON: Knowledge representation meets information retrieval. In *Proceedings of the 1st Conference on Artificial Intelligence Applications* (*CAIA '84*). IEEE Computer Society Press, Los Alamitos, Calif., 280–286.

POULIN, J. S. AND CARUSO, J. M. 1993. A reuse metrics and return on investment model. In *Advances in Software Reuse*. IEEE Computer Society Press, Los Alamitos, Calif., 152–166.

PRIETO-DÍAZ, R. 1985. A software classification scheme. Ph.D. dissertation, Tech. Rep. 85-19, Univ. of California—Irvine, Irvine, Calif.

PRIETO-DÍAZ, R. 1991. Implementing faceted classification for software reuse. *Commun. ACM 35*, 5 (May).

PRIETO-DÍAZ, R. AND ARANGO, G. 1991. *Domain Analysis and Software Systems Modeling*. IEEE Computer Society Press, Los Alamitos, Calif.

PRIETO-DÍAZ, R. AND FREEMAN, P. 1987. Classifying software for reusability. *IEEE Softw. 4*, 1, 6–16.

ROSE, D. E. AND BELEW, R. K. 1991. A connectionist and symbolic hybrid for improving legal research. *Int. J. Man Mach. Stud. 35*, 1, 1–33.

SALTON, G. AND BUCKLEY, C. 1988. Term weighting approaches in automatic text retrieval. *Inf. Process. Manage. 24*, 3, 513–525.

SALTON, G. AND BUCKLEY, C. 1990. Improving retrieval performance by relevance feedback. *J. Am. Soc. Inf. Sci. 41*, 4, 288–297.

SALTON, G. AND MCGILL, M. J. 1983. *Introduction to Modern Information Retrieval*. McGraw Hill, New York.

SHAW, M. AND GARLAN, D. 1996. *Software Architecture: Perspectives on an Emerging Domain*. Prentice-Hall, Englewood Cliffs, N.J.

SIMOS, M., CREPS, D., KLINGER, C., AND LEVINE, L. 1995. Organization domain modeling (ODM) guidebook. STARS-VC-A023/011/00, Unisys Corp., Reston, Va.

SMOLENSKY, P. 1988. On the proper treatment of connectionism. *Behav. Brain Sci. 11*, 1–23.

SOMMERVILLE, I. AND WOOD, M. 1986. A software components catalogue. In *Intelligent Information Systems: Progress and Prospects*. R. Davies, Ed. Ellis Horwood Limited, Chichester, U.K., 13–32.

SPARCK-JONES, K. 1972. A statistical interpretation of term specificity and its application in retrieval. *J. Doc. 28*, 1, 11–21.

SPARCK-JONES, K. 1973. Index term weighting. *Inf. Storage Retriev. 9*, 619–633.

STALLMAN, R. M. 1981. EMACS, the Extensible, Customizable, Self-Documenting Display Editor. *ACM SIGOA Newslett. 1*, 1/2, 147–156.

THOMPSON, R. H. AND CROFT, W. B. 1989. Support for browsing in an intelligent text retrieval system. *Int. J. Man Mach. Stud. 30*, 639–668.

WALZ, D. B., ELAM, J. J., AND CURTIS, B. 1993. Inside a software design team: Knowledge acquisition, sharing, and integration. *Commun. ACM 36*, 10 (Oct.), 62–77.

WIEDERHOLD, G., WEGNER, P., AND CERI, S. 1992. Toward megaprogramming. *Commun. ACM 35*, 11 (Nov.), 89–99.

WILLIAMS, M. D. 1984. What makes RABBIT run? *Int. J. Man Mach. Stud. 21*, 333–352.

WIRFS-BROCK, R. J. AND JOHNSON, R. E. 1990. Surveying current research in object-oriented design. *Commun. ACM 33*, 9 (Sept.), 105–124.

Received July 1995; revised February 1996; accepted November 1996