Context-Aware Browsing of Large Component Repositories

Yunwen Ye[1,2]
[1]SRA Key Technology Laboratory, Inc.
3-12 Yotsuya, Shinjuku
Tokyo, 160-004, Japan
yunwen@cs.colorado.edu

Gerhard Fischer[2]
[2]Department of Computer Science
CB430, University of Colorado
Boulder, CO80309-0430, USA
gerhard@cs.colorado.edu

Abstract

This paper proposes a new approach to locating software components from a large component repository: context-aware browsing. Without any explicit input from software developers, this approach automatically locates and presents a list of software components that could possibly be used in the current development situation. This automation of the component location process not only greatly reduces the search space of components so that software developers can easily browse and choose the desired components, but also enables software developers to use components whose existence they do not even anticipate. A software agent that supports context-aware browsing has been developed and evaluated.

1 Introduction

Component-based software development improves both the quality and productivity of software development. However, before software developers can use or reuse a software component (e.g., a class or a method), they must have learned it beforehand or be able to locate it from a component repository. Component repositories are often so large that software developers cannot learn about all of the components. Moreover, component repositories are not static; they are constantly evolving as new components are added and old components updated. As an example, Table 1 shows the rapid growth of the Java Core API (Application Programmer Interface) library. Few Java developers, if any, can claim that they know all the components in this library. To use components they do not yet know, software developers need to locate them from component repositories.

Version	No. of packages	No. of classes	Year of release
Java 1.0	8	211	1996
Java 1.1	23	503	1997
Java 1.2	59	1525	1998
Java 2	70+	2100+	1999

Table 1: Growth of the Java API library

Most of the research on component repository systems that support component location has focused on improving either browsing or searching mechanism. This paper describes a software agent, *CodeBroker*, that explores a new approach to locating software components: context-aware browsing. Running continuously in the background of a development environment, *CodeBroker* infers software developers' needs for components by monitoring their interactions with the development environment. Potentially reusable components that match the development context—the development task on which developers are working and the background knowledge of developers—are autonomously located and actively presented. Although the autonomously located components may not be precise enough, the number of components that developers have to browse is greatly reduced.

Empirical evaluations have found that *CodeBroker* can expedite component location. More important, it helps software developers discover those components whose existence is not anticipated, and thus increases the opportunity of component reuse.

2 Searching and Browsing

Searching and browsing have long served as the principal techniques for software developers to locate components from component repositories.

Searching is direct and fast. Software developers formulate a query, and the repository system returns components that match the query. Formulating queries is a cognitively challenging task because software developers have to overcome the gap from the situational model (i.e., the software developers' understanding of their task) to the system model (i.e., the description of the components in the repository) [25].

In browsing, software developers determine the usefulness or relevance of the components currently being displayed in terms of their development task, and traverse the associated links in the component repository. In general, people who look for information prefer browsing

to searching because they do not need to commit resources at first and can incrementally develop their requirements after evaluating the information along the way [22]. Mili et al. [17] claim that browsing is the predominant pattern of component repository usage because many software developers often cannot formulate clearly defined queries. Instead, they rely on browsing to get acquainted with available components in the repository.

Browsing, however, is not scalable for the following reasons. First, there is an inherent dilemma in the design of the browsing structure: If links are too many, software developers will be puzzled by the complexity; if links are too few, components are not well connected. Second, it is impossible to design a structure suitable for all developers and all tasks. Most component repositories are structured according to the inheritance relationship. This structure is suitable for locating components whose super nodes (classes or super-classes) are known, but it is not suitable for locating components based on functionality. Some components with similar functionality are scattered in different deep nodes of the inheritance tree [12]. It is very difficult for software developers to find and compare all of them in order to choose the most appropriate one. Third, in a large component repository, following the right link requires that software developers have a very good understanding of the structure of the whole repository. Most software developers, especially the less experienced ones, may easily get lost in a complex network of nodes while tracing dozens of links.

3 Beyond Searching and Browsing

3.1 Information Access and Information Delivery

Searching and browsing are information access mechanisms that require information users (software developers in our case) to initiate the information-seeking process. Consequently, information access mechanisms offer no support for those users who do not ask for new information. To assist users in making full use of large information repositories, information access mechanisms need to be complemented with information delivery mechanisms that actively present information to users without being given explicit queries [8]. The "Tips of the Day" of the Microsoft Windows system, which voluntarily presents a tip on the use of an application, is an example of the information delivery mechanism.

Incorporating the information delivery mechanism with a component repository can encourage software developers who make no attempt to take advantage of components. Many software developers create their own programs instead of attempting to use components from the repository simply because they do not know the existence of components or they do not know how to find the components [10, 20]. This phenomenon is well illustrated in the following comment from a software developer.

> "I could be creating a method that does exactly the same thing somebody else's does…even though we have access to each other's code. We might call them different names and we might have a bit different way of doing it, but we're still doing the same thing." [5]

3.2 Context-Aware Browsing

Information delivery systems that just throw a piece of decontextualized information at users are of little use because they ignore the working context. The working context consists of the task being performed and the user performing it. The challenge for information delivery is to present context-aware information related to both the task at hand and the background knowledge of the user [7].

The context-aware browsing mechanism strives to deliver contextualized components. Instead of asking software developers to pose queries, it infers their tasks at hand by continuously monitoring their interactions with the development environment, and it autonomously locates and delivers a list of components that could be used in the implementation of the tasks and that are not yet known to the developers. Because the tasks are inferred, the list of delivered components may not be accurate enough and may include some irrelevant components. Software developers will still need to browse the delivery to find the components they want, but compared to the entire repository, the browsing space is significantly smaller and the time to find the desired components is thus reduced. The delivered components are the result of a first-cut search [18] automated by component repository systems.

Table 2 compares context-aware browsing with browsing and searching by summarizing their advantages and disadvantages.

	Advantages	**Disadvantages**
Browsing	Low cognitive overheads	Does not scale up
Searching	Fast, direct	Formulating the "right" query is difficult No search for unanticipated components
Context-aware browsing	Supports information delivery	Difficulty in "understanding" the context

Table 2: Comparison of locating mechanisms

4 A Software Agent that Supports Context-Aware Browsing

We have developed and evaluated an autonomous software agent—*CodeBroker*—that supports context-aware browsing. *CodeBroker*, whose architecture is shown

in Fig. 1, supports Java developers. Its component repository currently includes 673 classes and 7,338 methods from JDK (Java Development Kit) 1.1.8 Core API library and JGL 1.3 (Java General Library, created and distributed by ObjectSpace, Inc.). Other components can easily be added with the indexing program we have developed [25].

CodeBroker consists of three subsystems: *Listener*, *Fetcher*, and *Presenter*. *Listener*, running continuously behind the program editor—Emacs, extracts and formulates queries by monitoring the software developers' activities. Those queries are passed to *Fetcher*, which retrieves matching components. Retrieved components are delivered in the *Reusable Components Information Display (RCI-display)* by *Presenter*, after it has removed unwanted components based on the background knowledge of the targeted developer.

CodeBroker is integrated with the editor. A portion of the editor is used for the display of delivered components (*RCI-display*). The integration reduces the unnecessary switch between development activities and component search activities, which causes the loss of working memory. Because delivered components are immediately accessible from the current development environment, software developers are able to skim or perceive components at a glance instead of going through a lengthy browsing or searching process.

4.1 Inferring the Needs for Components

CodeBroker utilizes information contained in unfinished programs in the editor to infer software developers' needs for components. A program has three aspects: concept, code, and constraint. The concept of a program is its functional purpose, the code is the embodiment of the concept, and the constraint is the environment in which it runs [26].

Important concepts of a program are often revealed by its informal information. Programs include both formal information for executability and informal information for readability. Informal information includes structural indentation, comments, and identifier names. Because comments and identifier names often reveal the semantics of programs, they not only increase the comprehensibility of programs, but also serve as cues signaling the needs for components that can be used in their implementation. The Java programming language further stresses the importance of comments by introducing the concept of doc comments. A doc comment begins with "/**" and continues until the next "*/". It precedes the declaration of a module (either a class or a method), and its contents describe the functionality of the following module. Doc comments are extracted by the *javadoc* utility to create documentation for Java programs.

One important constraint of a program is its type compatibility, which is manifested in its signature. A signature is the type expression of a program, and defines its syntactical interface. For a component to be easily integrated, its signature should be compatible with the environment into which it is going to be incorporated.

Combining the concept revealed by comments and identifier names, and constraints revealed by signatures, task-relevant components can be found. If the component shows high similarity in concept and high compatibility in

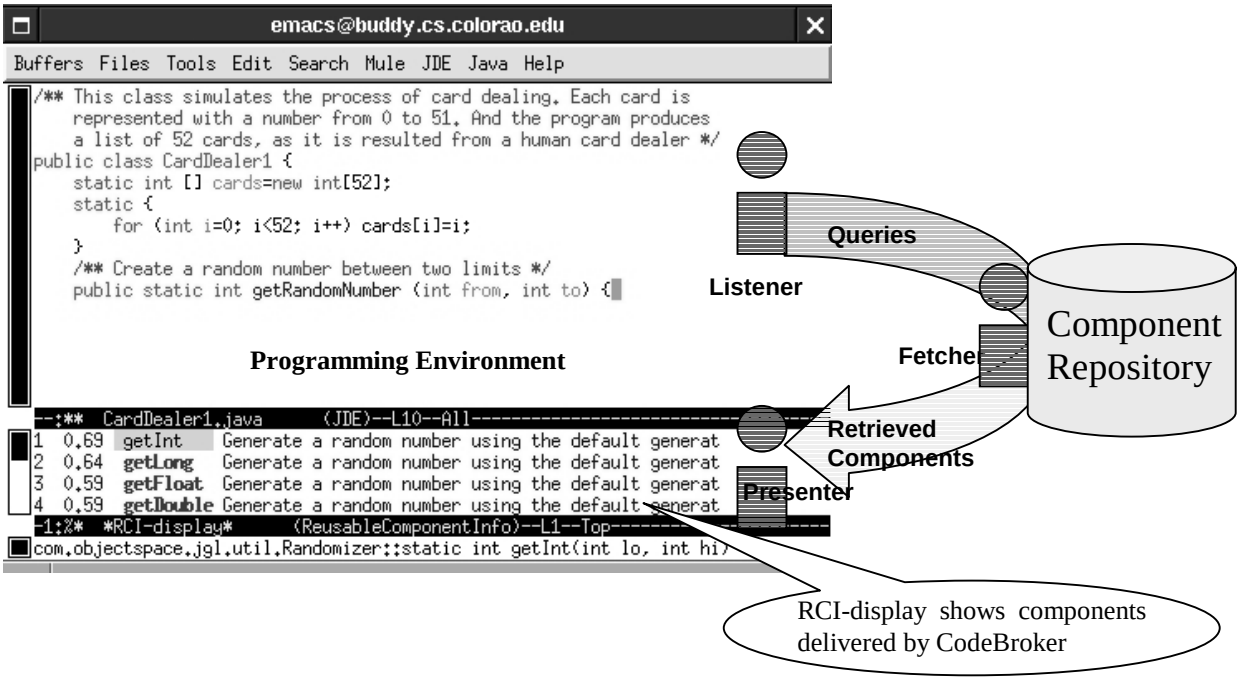

Figure 1: Overview of the *CodeBroker* agent

constraint, the likelihood of a component matching the developer's task is also high. *CodeBroker* infers software developers' needs for components by analyzing the contents of doc comments and signatures. As soon as a doc comment is entered in the editor, *CodeBroker* begins to search for and deliver components whose documents are similar to the comment. A doc comment for a method is followed by the declaration of the signature for the same method. After the method signature is declared in the editor, *CodeBroker* makes a second attempt to deliver components that takes both the signature and doc comment into consideration in judging the task-relevance of components. For example, in Fig. 1, components shown in *RCI-display* are delivered based on their relevance to the doc comment and the signature (the two lines before the cursor); and the first component does exactly what the developer wants to do.

4.2 Retrieving and Delivering Relevant Components

CodeBroker uses both free-text information retrieval techniques and signature matching to retrieve task-relevant components.

It uses the probability-based information retrieval technique, proposed by Robertson and Walker [19], to compute the concept similarity—the similarity between conceptual queries extracted from doc comments of programs under development and documents of components in the repository. Based on the assumption that terms are distributed differently in relevant and irrelevant documents, the probability-based information retrieval technique estimates the similarity between a query and a document by assigning appropriate weights to terms in the document collection, and returns a rank-ordered list of pre-indexed documents that best match the query. The concept similarity between a query (Q) and the document of a component (D_j, $0<j<N$, where N is the number of components in the repository) is computed as follows:

$$sim(Q, D_j) = \sum_{i=1}^{T} (\log \frac{N - n_i + 0.5}{n_i + 0.5}) \frac{(k_1+1)tf_{i,j}}{K + tf_{i,j}} \frac{(k_3+1)qtf_i}{k_3 + qtf_i}$$

where

N is the number of components
n_i is the number of components whose documents contain the term t_i
T is the number of terms in the component collection
$tf_{i,j}$ is the frequency of term t_i in the document of the component D_j
qtf_i is the frequency of term t_i in the query Q
$K = k_1((1-b) + b \cdot dl_j / avdl$
k_1,k_3,b are empirically determined parameters depending on the nature of the document collection. In *CodeBroker*, k_1 is set to 1.2, k_3 to 1.0, and b to 0.75, according to the data in [23].
dl_j is the length of document D_j
$avdl$ is the average length of all documents in the collection

Signature matching is used to determine the constraint compatibility—the compatibility between constraint queries extracted from signatures of programs being developed and signatures of components in the repository [27]. The basic form of a signature of a method is

```
Signature:InTypeExp->OutTypeExp
```

where `InTypeExp` and `OutTypeExp` are type expressions that result from applying a Cartesian product constructor onto the input parameter types and output parameter types, respectively. The two signatures

```
Sig1:InTypeExp1->OutTypeExp1
Sig2:InTypeExp2->OutTypeExp2
```

match if and only if `InTypeExp1` is in structural conformance with `InTypeExp2` and `OutTypeExp1` is in structural conformance with `OutTypeExp2`. Two type expressions are structurally conformable if they are formed by applying the same type constructor to structurally conformant types. The constraint compatibility value between two signatures is the product of the conformance value existing among their types. The type conformance value is 1.0 if two types are in structural conformance according to the definition of the programming language, and drops a certain percentage if one type conversion is needed, or an immediate inheritance relationship exists between them [26].

4.3 The Delivery-Browsing-Searching Cycle

The top 20 components (the number of delivered components can be easily customized by users) with the highest similarity values are presented in *RCI-display* in decreasing order of similarity value. Each delivered component is accompanied with its rank of similarity, similarity value, name, and a short description (Fig. 1).

Delivered components can be treated as the results of information reconnaissance [14] conducted by *CodeBroker*. During this process, the system explores unknown territory of the component repository before the software developers are committed to entering it. Software developers can take advantage of the reconnaissance results in several ways.

- They can quickly browse the list and continue programming if they do not find anything interesting. This could reduce the lost time caused by unsuccessful efforts at locating components from the whole repository.
- They can use a component immediately if they find it useful and the information present in *RCI-display* is adequate.
- They can launch, by left-clicking the component name, an external HTML browser to go to the corresponding place for the full component documents generated by *javadoc*, if they find some interesting components but need more information.
- If too many irrelevant components are delivered, they can invoke the `Skip Components Menu` (Fig. 2) associated with each delivered component by right-clicking it. A method component (the first item in the

menu), or all methods from a class (the second item) or a package (the third item) can be removed from *RCI-display* to make it easier to find the desired components.

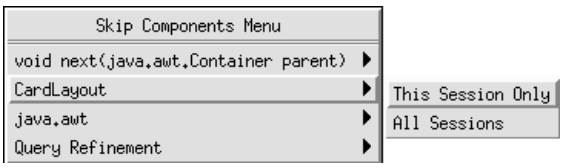

Figure 2: The Skip Components Menu

- They can start another round of searching by activating the *retrieval-by-reformulation* [24] interface (Fig. 3) to reformulate the query. In such a scenario, the delivered components serve the role of acquainting software developers with the vocabulary used in the repository system so that they can write a more appropriate query. By analyzing the delivered components, software developers become familiar with the structure of the repository, and can decide to limit the search on a particular portion of the repository by specifying the `Interested Components` field (which takes packages or classes as input and ensures that only components from specified packages and classes are retrieved), or exclude a particular portion of no interest by specifying the `Filtered Components` field (which ensures that components from the specified packages and classes are not retrieved) (Fig. 3)

Figure 3: The retrieval-by-reformulation interface

4.4 Capturing the Larger Context

The delivery mechanism described in the preceding sections infers the need for components from the module on which a software developer is currently working. A module is only a part of the whole development task, and the functionality of the module is embedded in a larger context shaped by other modules that have been developed so far. Therefore, the interactions between the developer and *CodeBroker* that have taken place for the development of previous modules can be used to improve the context-relevance of later deliveries. In *CodeBroker*, such an interaction history is represented as a discourse model that is used to adapt the delivery of components to each development session.

Component repositories are often organized hierarchically according to packages and classes that are designed for particular application domains. For most applications, only a part of the repository is involved. If the repository system knows which part of the repository is relevant, the context-relevance of located components can be improved. A specification mechanism [9] may be used to allow software developers to specify all of their interested packages and classes before the development session starts, but it is difficult for most software developers to do so because they may not have enough knowledge about the whole repository. It is much easier for them to identify a definitely irrelevant package or class when they see one. Based on the above observation, discourse models in *CodeBroker* include those packages, classes, and methods that are not relevant to the current development session, and components contained in discourse models are not delivered, even if they are relevant to the queries. Discourse models also reduce the delivery of irrelevant components caused by polysemy—a difficult problem for any information retrieval systems—by limiting search domains because polysemous words often have different meanings in different domains.

Each development session starts with an empty discourse model, and the contents of discourse models are incrementally added during the interaction between developers and the system. When developers invoke the `Skip Components Menu` (Fig. 2), they can choose to remove a method, a class, or a package from the *RCI-display* for the current development session by choosing the `This Session Only` command. At the same time, *CodeBroker* learns that those components are not of interest to the developers at this development session and adds them to the discourse model.

4.5 Adapting to Each Developer

As mentioned in Section 3.2, another element of the context is the software developer's knowledge of components. Presenting a component that a software developer already knows very well is of little use because if it can be used in the current development, the developer would be able to do so without help. *CodeBroker* creates a user model for each developer by analyzing the programs that the developer has written. The user model contains components that have been used by the developer more than three times (this number is adjustable by the developer). We assume that if a developer has used a component more than three times, he or she knows it already; therefore, it should not be presented by *CodeBroker*, even if it is determined to be relevant to the queries by the retrieval mechanisms.

Because software developers' knowledge of components evolves as time goes by, user models should evolve accordingly. *CodeBroker* provides a mechanism for software developers to explicitly update their own user

models, as well as a mechanism that updates user models implicitly by analyzing the interactions between software developers and the system.

Similar to updating of discourse models, software developers can update their user models explicitly by choosing the `All Sessions` command in the `Skip Components Menu` (Fig. 2). The difference between user models and discourse models are that user models are stored in permanent storages and are loaded into the system each time it is started, whereas discourse models are stored in the memory and are re-initialized to empty when the system is started. A user model is the shared long-term memory between *CodeBroker* and a developer; in contrast, a discourse model is the shared short-term memory.

CodeBroker continuously learns what software developers know about the repository components and updates their user models accordingly by monitoring their development activities. When the system observes that a particular component has been used by a software developer more than three times, it adds the component to his or her user model. For more details on the algorithm of automatically updating user models, see reference [25].

5 Evaluations of *CodeBroker*

We have conducted two types of experiments to evaluate *CodeBroker*. The first experiment evaluates the retrieval mechanism, and the second set of experiments empirically studies how well *CodeBroker* assists software developers in discovering components during their development activities.

5.1 Retrieval Effectiveness

Information retrieval systems are conventionally measured by recall and precision [21]. Recall—the percentage of the total relevant documents in the collection that are also retrieved—indicates the ability of the system to retrieve all relevant documents. Precision—the percentage of documents retrieved that are relevant—indicates the ability of the system to present only relevant documents.

Recall	Average Precision
0%	45.82%
10%	45.82%
20%	45.82%
30%	41.20%
40%	41.01%
50%	40.74%
60%	37.46%
70%	37.46%
80%	32.71%
90%	32.19%
100%	29.43%

Table 3: Recall and precision of *CodeBroker*

We tried 19 queries in the experiment [25]. Among them, 10 queries were created by us, 4 were chosen from questions frequently asked in newsgroups for Java programming, and 5 were extracted from the empirical evaluation experiments. Table 3 shows the average precision and recall values. In average, about one-third to one-half of the retrieved components are relevant to the development task, which means that one out of two or three delivered components can be used to implement the module described by the queries.

5.2 Empirical Evaluations

Recall and precision are not absolute, objective measurements of an information retrieval systems because (1) the definition of relevance between documents and queries is subjective, and (2) even if the relevance of a document is unanimously agreed, it may not be of interest to one particular user if that user already knows the document. To better understand how *CodeBroker* supports software developers during their development practice, we have conducted 12 experiments with five subjects. All subjects had extensive programming experience, and their expertise in Java programming varied from medium to expert. In each experiment, the subject was asked to implement a small programming task that could be developed with different components from the repository if the subject knew or found them.

During the experiments, we logged all the components that were retrieved and delivered by *CodeBroker* and those that were used in the final programs. The analysis was based on the logs and transcribed protocols of the experiments and follow-up interviews. Table 4 summarizes the overall results of these experiments.

In 10 of the 12 experiments, the subjects used components delivered by *CodeBroker*. The 12 programs created by the subjects used 57 distinct components, 20 of which were delivered by *CodeBroker*. Of these 20 used components, the subjects did not anticipate the existence of 9. In other words, those 9 components could not have been used without the support of *CodeBroker*, and the subjects would have created their own solutions instead. Although the subjects somehow anticipated the existence of the other 11 components, they had known neither the names nor the functionality, and had never used them before. They might have used those 11 components if they could manage to locate them by themselves. In follow-up interviews, all subjects acknowledged that *CodeBroker* made locating those anticipated components much easier and faster. The last column shows the rates given by the subjects when they were asked to rate the usefulness of the system on a scale from 1 (totally useless) to 10 (extremely

Subject	Experiment no.	Total no. of distinct components used	No. of distinct components used from deliveries	Breakdown of used components from deliveries		Rate of the system (0: worst 10: best)
				No. of components whose existence was unanticipated	No. of components whose existence was anticipated but unknown	
S1	1	10	4	2	2	7
	2	3	1	1	0	
S2	3	7	1	1	0	4
	4	4	1	1	0	
	5	5	3	0	3	
S3	6	5	2	1	1	8.5
	7	4	3	1	2	
	8	3	0	0	0	
S4	9	4	3	0	3	7
	10	3	1	1	0	
S5	11	4	1	1	0	8
	12	5	0	0	0	
SUM		57	20	9	11	

useful). Although the rates are subjective, they indicate the subjects' desire to use the system continuously.

6 Related Work

Component repository systems that support searching have adopted different retrieval mechanisms. Based on the representation schema of components, they can be divided into text-based, structured representation-based, and formal method-based approaches.

The text-based approach uses textual descriptions to represent components and queries, and adopts information retrieval techniques to define the task-relevance of components. Textual descriptions are drawn from accompanying documents [15], or are extracted from comments and/or identifier names in components [4]. Structured representation-based approaches employ a knowledge base or conceptual distance graph to mimic the method a human being would employ in searching components. Multi-faceted classification schemas [2] use multiple facets to represent components and a conceptual distance graph to reflect the semantic relationship among terms describing components. Both *CodeFinder* [13] and *LaSSIE* [1] represent components as frames. *CodeFinder* organizes those frames into an associated network and uses spreading activation to find components. Frames in *LaSSIE* are organized into hierarchical, taxonomic categories. Formal method-based approaches use either signatures [27] or formal specifications [16] to represent components and queries.

Most component repository systems that support browsing, such as the Smalltalk programming environment, organize components according to their inheritance structure. Such a browsing structure requires that software developers have extensive knowledge about the structure of the repository to find a component, and it lacks meaningful road maps as to what the repository contains and how to discover the components that are needed for the current task [11].

Several attempts have been made to improve the browsing of a large component repository. Fischer [6] proposes a specification-based browsing mechanism that combines concepts and formal specifications to structure the repository. This mechanism enables software developers to find components by selecting features required in their task, or deselecting features not needed for the task. Drummond et. al [3] add an active agent to the existing browsing system that speeds up the search for components. The agent infers the search goal of software developers by observing their browsing actions and delivers components that closely match the inferred goal.

Compared to other repository systems, our approach is unique in the following aspects:

(1) The first attempt to locating components is automated by the system.
(2) The system adapts the search process to each development session.
(3) The system adapts the search process to the background knowledge of each developer.

7 Conclusions

This paper has described the idea of context-aware browsing to assist software developers in discovering components from a large repository. Context-aware browsing utilizes information delivery techniques to

deliver a list of contextualized components without requiring direct operations from software developers; it combines the benefits of the low cognitive threshold of browsing and the directness of searching.

We have designed and implemented *CodeBroker* to demonstrate the feasibility of supporting context-aware browsing. Empirical evaluations of the system have shown that context-aware browsing not only speeds up the search for components, but also enables software developers to learn and use unanticipated components that they might miss with the conventional information access mechanisms of browsing and searching, because information access mechanisms can help only those software developers who anticipate the existence of components and initiate the component locating process.

Acknowledgments. The authors thank the members of the Center for LifeLong Learning & Design at the University of Colorado, who have made major contributions to the conceptual frameworks described in this paper. The research was supported by (1) the National Science Foundation, Grant REC-0106976; (2) SRA Key Technology Laboratory, Inc., Tokyo, Japan; and (3) the Coleman Family Foundation, San Jose, CA.

8 References

[1] Devanbu, P., *et al.*, LaSSIE: A Knowledge-Based Software Information System. *Communications of the ACM*. **34**(5):34-49, 1991.

[2] Diaz, R.P. and P. Freeman, Classifying Software for Reusability. *IEEE Software*. **4**(1):6-16, 1987.

[3] Drummond, C., D. Ionescu, and R. Holte, A Learning Agent that Assists the Browsing of Software Libraries. *IEEE Transactions on Software Engineering*. **26**(12):1179-1196, 2000.

[4] Etzkorn, L.H. and C.G. Davis, Automatically Identifying Reusable OO Legacy Code. *IEEE Computer*. **30**(10):66-71, 1997.

[5] Fichman, R.G. and C.E. Kemerer, Object Technology and Reuse: Lessons from Early Adopters. *IEEE Software*. **14**(10):47-59, 1997.

[6] Fischer, B., Specification-Based Browsing of Software Component Libraries. *Automated Software Engineering*. **7**(2):179-200, 2000.

[7] Fischer, G., Articulating the Task at Hand and Making Information Relevant to It. *Human-Computer Interaction*. (to appear), 2001.

[8] Fischer, G., Domain-Oriented Design Environments. *Automated Software Engineering*. **1**(2):177-203, 1994.

[9] Fischer, G., K. Nakakoji, and J. Ostwald, Supporting the Evolution of Design Artifacts with Representations of Context and Intent. in *Proc. of Designing Interactive Systems '95*, (New York), 1995, 7-15.

[10] Frakes, W.B. and C.J. Fox, Quality Improvement Using a Software Reuse Failure Modes Model. *IEEE Transactions on Software Engineering*. **22**(4):274-279, 1996.

[11] Green, T.R.G., Programming Languages as Information Structures, in *Psychology of Programming*, J.-M. Hoc, *et al.*, (eds.) Academic Press: New York. 118-137, 1990.

[12] Helm, R. and Y.S. Maarek, Integrating Information Retrieval and Domain Specific Approaches for Browsing and Retrieval in Object-Oriented Class Libraries. in *Proc. of OOPSLA'91*, 1991, 47-61.

[13] Henninger, S., An Evolutionary Approach to Constructing Effective Software Reuse Repositories. *ACM Transactions on Software Engineering and Methodology*. **6**(2):111-140, 1997.

[14] Lieberman, H., Personal Assistants for the Web: An MIT Perspective, in *Intelligent Information Agents: Agent-Based Information Discovery and Management on the Internet*, M. Klusch, (ed.) Springer-Verlag: Berlin. 279-292, 1999.

[15] Maarek, Y.S., D.M. Berry, and G.E. Kaiser, An Information Retrieval Approach for Automatically Constructing Software Libraries. *IEEE Transactions on Software Engineering*. **17**(8):800-813, 1991.

[16] Mili, A., R. Mili, and R. Mittermeir, Storing and Retrieving Software Components: A Refinement-Based System. *IEEE Transaction on Software Engineering*. **23**(7):445-460, 1997.

[17] Mili, A., *et al.*, Toward an Engineering Discipline of Software Reuse. *IEEE Software*. **16**(5):22-31, 1999.

[18] Mili, H., F. Mili, and A. Mili, Reusing Software: Issues and Research Directions. *IEEE Transactions on Software Engineering*. **21**(6):528-562, 1995.

[19] Robertson, S.E. and S. Walker, Some Simple Effective Approximations to the 2-Poisson Model for Probabilistic Weighted Retrieval. in *Proc. of the 17th International ACM-SIGIR Conference*, (Dublin, Ireland), 1994, 232-241.

[20] Rosenbaum, S. and B. DuCastel, Managing Software Reuse--An Experience Report. in *Proc. of 17th International Conference on Software Engineering*, (Seattle, WA), 1995, 105-111.

[21] Salton, G. and M.J. McGill, *Introduction to Modern Information Retrieval*, McGraw-Hill: New York. 1983.

[22] Thompson, R.H. and W.B. Croft, Support for Browsing in an Intelligent Text Retrieval System. *International Journal of Man-Machine Studies*. **30**(6):639-668, 1989.

[23] Walker, S., *et al.*, Okapi at TREC-6: Automatic ad hoc, VLC, Routing, Filtering and QSDR. in *Proc. of the 6th Text REtrieval Conference (TREC-6)*, (Gaithersburg, MD), 1998, 125-136.

[24] Williams, M., What Makes RABBIT Run? *International Journal of Man-Machine Studies*. **21**:333-352, 1984.

[25] Ye, Y., *Supporting Component-Based Software Development with Active Component Repository Systems*. Ph.D. Dissertation, Department of Computer Science, University of Colorado, Boulder, CO, 2001.

[26] Ye, Y. and G. Fischer, Promoting Reuse with Active Reuse Repository Systems. in *Proc. of the 6th International Conference on Software Reuse*, (Vienna, Austria), 2000, 302-317.

[27] Zaremski, A.M. and J.M. Wing, Signature Matching: A Tool for Using Software Libraries. *ACM Transactions on Software Engineering and Methodology*. **4**(2):146-170, 1995.