Reuse-Conducive Development Environments

Yunwen Ye[1,2]

[1]SRA Key Technology Laboratory, Inc.
3-12 Yotsuya, Shinjuku
Tokyo 160-004, Japan
+81-03-3357-9361
yunwen@cs.colorado.edu

Gerhard Fischer[2]

[2]Department of Computer Science
University of Colorado
Boulder, CO 80303-0430, USA
+1-303-492-1592
gerhard@cs.colorado.edu

ABSTRACT

Despite its well-recognized benefits, software reuse has not met its expected success due to technical, cognitive, and social difficulties. We have systematically analyzed the reuse problem (especially the cognitive and social difficulties faced by software developers who reuse) from a multidimensional perspective, drawing on our long-term research on information retrieval, human-computer interaction, and knowledge-based systems. Based on this analysis, we propose the concept of *reuse-conducive development environments*, which encourage and enable software developers to reuse through the smooth integration of reuse repository systems and development environments. We have designed, implemented, and evaluated *CodeBroker*—a reuse-conducive development environment—that autonomously locates and delivers task-relevant and personalized components into the current software development environment. Empirical evaluations of *CodeBroker* have shown that the system is effective in promoting reuse by enabling software developers to reuse components unknown to them, reducing the difficulties in locating components, and augmenting the programming capability of software developers.

KEYWORDS

Software reuse; reuse-conducive environments; high-functionality applications; information delivery; relevance to the task-at-hand; personalization; location, comprehension, and modification model; latent semantic analysis; centralized and decentralized development of reuse repositories; seeding, evolutionary growth, and reseeding model; CodeBroker

Table of Contents

1. INTRODUCTION 4
2. THE CONCEPTUAL FRAMEWORK 5
2.1 Cognitive Issues in Reuse 6
2.2 Information Retrieval and Reuse 7
2.3 Knowledge-Based Systems and Reuse 8
3. CODEBROKER: DELIVERING TASK-RELEVANT AND PERSONALIZED COMPONENTS 10
3.1 Overview of the CodeBroker System 11
3.2 Locating Task-Relevant Components 13
3.3 Supporting Retrieval-by-Reformulation 15
3.4 Creating and Using Discourse Models 16
3.5 Delivering Personalized Components 19
4. EVALUATION 20
4.1 Recall and Precision 21
4.2 The Structure of the Experiments 22
4.3 Findings of Experiments 23
5. DISCUSSION 28
6. EVOLUTIONARY CONSTRUCTION OF REUSE REPOSITORIES 29
6.1 The Centralized Paradigm 30
6.2 The Decentralized, Evolutionary Paradigm 30
6.3 A Comparison of the Two Paradigms 32
7. RELATED WORK 33
7.1 Information Delivery Systems 33
7.2 Reuse Repository Systems 35
8. SUMMARY 37
9. REFERENCES 38

List of Figures:

Figure 1: Different levels of knowledge about a high-functionality application (HFA) 5
Figure 2: Similarity analysis 9
Figure 3: An example of the use of CodeBroker 10
Figure 4: The system architecture of CodeBroker 11
Figure 5: The retrieval-by-reformulation interface 15
Figure 6: An example discourse model 17
Figure 7: Updating the discourse model 17
Figure 8: Deliveries with and without the discourse model 18
Figure 10: The recall-precision curve 21

Figure 11: The Seeding, Evolutionary Growth, and Reseeding (SER) model _____ *31*
Figure 12: Two general paradigms of creating and using an information repository: (a) centralized, and (b) decentralized. _____ *33*

List of Tables:

Table 1: Programming knowledge and expertise of subjects _____ *23*
Table 2: Overall results of empirical evaluations _____ *24*

1. INTRODUCTION

Although many believe that software reuse improves both the quality and productivity of software development (Basili, Briand et al., 1996), systematic reuse has not yet met its expected success. Instituting a reuse program involves two essential issues:

- Creating and maintaining a reuse repository, which requires managerial commitments and substantial investments, both financially and intellectually;
- Enabling software developers to build new software systems with components from the reuse repository.

These two issues are in a *deadlock*: if software developers are unable to reuse, the investments in reuse cannot be justified; conversely, if companies are unwilling to invest in reuse, software developers have little to reuse. One approach to breaking this deadlock is to focus first on the creation of a good reuse repository and then to institute a reuse program top-down by enforcing reuse through education and other organizational changes (Fafchamps, 1994). A second approach is to foster a reuse culture bottom-up by encouraging software developers to reuse components from a repository that may not be of high quality in its initial state, but can be evolved through the participation and contribution of software developers (Henninger, 1997). Such an approach requires *reuse-conducive development environments* that not only enable but also encourage software developers to reuse existing components and contribute to the creation and evolution of reuse repositories.

Creating reuse-conducive development environments poses technical challenges—such as developing effective retrieval tools to help software developers locate, comprehend, and modify components (Fischer, Henninger et al., 1991)—as well as cognitive and social challenges—such as determining what triggers software developers to initiate the reuse process and what motivates them to contribute to the reuse repository. In this paper, we analyze and address the major cognitive challenge in software reuse: Software developers are unable, or unwilling, to initiate reuse if they do not have sufficient knowledge about the existence of reusable components or do not know how to locate them. Furthermore, we describe our ongoing research on creating a theoretical framework for evolving reuse repositories as well as our system-building efforts in supporting the evolution of reuse repositories through the active participation of software developers.

Most previous reuse research (A. Mili, R. Mili et al., 1998) has focused on designing various searching or browsing mechanisms to assist software developers in locating components. Both browsing and searching are passive mechanisms because they become useful only when software developers decide to make a reuse attempt by knowing or anticipating the existence of certain components. They are of little use for less experienced software developers who do not even anticipate the existence of components or do not know how to search the reuse repository properly. Modern reuse repositories contain not just 30 to 250 components, as Poulin (1999) claims, but thousands of

components; for example, the Java 1.4 API library has 2,723 classes. It is becoming increasingly difficult, therefore, for software developers to know or anticipate the existence of all available components. To enable the reuse of components whose existence software developers do not even anticipate and therefore cannot locate with traditional searching and browsing mechanisms, we propose a new style of interaction with reuse repository systems: *information delivery* that autonomously locates and presents to software developers task-relevant and personalized components without explicit reuse queries (Ye and Fischer, 2002). Our goal in this research is not to propose yet another component storage and retrieval mechanism, but to find a new way to interact with software reuse repository systems by means of enabling software developers to reuse without having to directly search the repository. We describe a system called *CodeBroker*, which illustrates different techniques to autonomously deliver components relevant to the task-at-hand and personalized to the background knowledge of an individual developer. Empirical evaluations of *CodeBroker* have shown that information delivery is effective in promoting reuse. Ongoing extension of the *CodeBroker* system to support and motivate software developers to participate actively in the creation and evolution of reuse repositories is also discussed.

2. THE CONCEPTUAL FRAMEWORK

Reuse repository systems are a subset of *high-functionality applications* (HFAs) (Fischer, 2001) that contain a large amount of information for computer users to access and use. The common problem faced by all HFAs is how to help users (or software developers in the case of reuse repository systems) locate, learn, and apply the task-relevant information that can help them accomplish a task-at-hand (Figure 1).

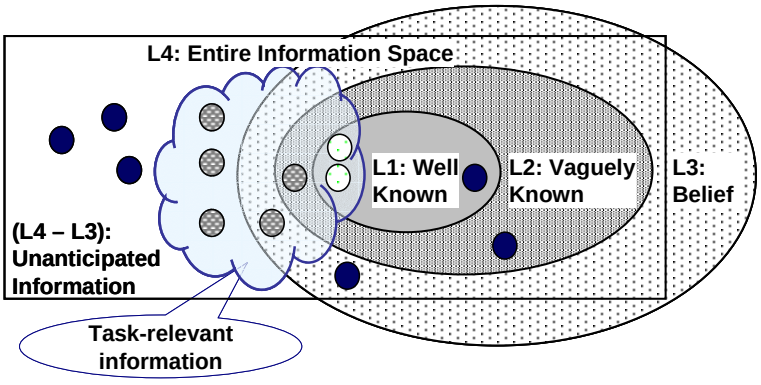


Figure 1: Different levels of knowledge about a high-functionality application (HFA)
The challenge in HFAs is how to differentiate task-relevant and personalized information from irrelevant information. The cloud represents the information needed for the *inferred* task-at-hand (with fuzzy boundaries because the system may have only a rudimentary understanding of it). The black dots are not relevant for the task-at-hand and should therefore not be delivered. The white dots inside the cloud should not be delivered because they are already known by a specific user (inferred from the user model, as discussed in Section 3.5).

Our empirical studies (Fischer, 2001) have shown that users typically have different levels of knowledge about HFAs. In Figure 1, the rectangle (L4) represents the actual information space, and the ovals (L1, L2, and L3) represent a particular user's level of knowledge of the information space: L1 represents the elements that are well known and can be easily used by the user, even without consulting help and documentation systems. L2 contains the elements that the user knows vaguely. L3 contains elements that the user anticipates to exist in the HFA. A portion of L3 lies outside the actual information space, so it contains elements that the user believes to exist, but they actually do not. The existence of many elements that fall in the area (L4 – L3) is not even anticipated by the user. Browsing and searching mechanisms that require users to initiate the information-locating process cannot help users obtain information in (L4 – L3) because users cannot ask for help if they are not even aware of the existence of available information.

We have long been concerned with designing both useful and usable HFAs in different application domains, and our research efforts enable us to examine the reuse problem from a multidimensional perspective. In this section, we discuss lessons we have learned from our research on human-computer interaction, information retrieval, and knowledge-based systems that have helped us create a conceptual framework for the software reuse problem.

2.1 Cognitive Issues in Reuse

The implication of Figure 1 for reuse is as follows: Because software developers know the components in L1 very well, they can reuse those components easily. Such a reuse approach is often referred to as "*opportunistic reuse*" (Sen, 1997) because its success relies solely on how much software developers know about components.

To achieve systematic reuse, software developers must be able to reuse not only the components they know, but also the components they do not yet know. Systematic reuse fails in the first place if software developers do not make an attempt to locate components. Such a phenomenon of "no attempt to reuse" (Frakes and Fox, 1996) is often regarded as an attitude problem, and is commonly labeled as the "Not-Invented-Here" syndrome. Many empirical studies (Lange and Moher, 1989; Frakes and Fox, 1995; Isoda, 1995) have shown, however, that software developers would put a lot of effort into locating and reusing components if they were aware of the components that could be reused. In other words, software developers are often very determined to reuse components in L2.

Reuse often fails not because software developers are unwilling to reuse, but because they are unable to do so due to the lack of appropriate knowledge about the operation of a reuse repository and its components. Much of the "Not-Invented-Here" phenomenon is caused by the cognitive difficulties that are inherent in the reuse process (Fischer, 1987a; Ye, Fischer et al., 2000; Ye and Fischer, 2002). That is,

- software developers may not have sufficient knowledge about the reuse repository and cannot even anticipate the existence of those components in the area (L4 – L3) that can be reused in their current task;

- software developers may perceive that reuse costs more than developing from scratch; and
- software developers may not be able to use the repository system by formulating a proper query or browsing the repository to locate components in L3.

2.2 Information Retrieval and Reuse

Most research on information retrieval is focused on designing an effective indexing and retrieval algorithm that achieves high recall and precision after users have formulated and submitted queries (Salton and McGill, 1983). Various schemas for indexing and retrieving software components have been proposed in previous reuse research (see Section 7.2). Although such schemas are very important, of equal (if not more) importance is investigating what motivates users to formulate queries as well as what kind of knowledge is needed for users to formulate queries.

Conceptual Gap between Situation Model and System Model. The needs for components are derived from development activities and are conceptualized in a *situation model*, which is the mental model software developers have of their development task (Kintsch, 1998). To locate components from a reuse repository, developers have to convert the situation model into the "actual" *system model*, which includes the ways of describing and structuring components in the repository. For example, a software developer who wants to draw a circle must know that the method is called `drawOval` in the Java class library in order to search for it, or must know that this method is in the `java.awt` package and in the `Graphics` class if he or she prefers browsing. This *conceptual gap* between situation and system models is a significant cognitive barrier to locating components (Fischer, Henninger et al., 1991). Two types of conceptual gap exist: *vocabulary mismatch* and *abstraction mismatch*. The vocabulary mismatch refers to the inherent ambiguity in most natural languages: People use a variety of words to refer to the same concept. The probability that two persons choose the same word to describe a concept is less than 20% (Furnas, Landauer et al., 1987; Harman, 1995). The abstraction mismatch refers to the difference of abstraction level in requirements and component descriptions. Programmers deal with concrete problems and thus tend to describe their requirements concretely; in contrast, reusable components are often described in abstract concepts because they are designed to be generic so they can be reused in many different situations (Ye, 2001).

Information Delivery. *Information delivery* ("push" technology) is a complementary approach to information access ("pull" technology), such as browsing and searching. Unlike information access, which requires users to initiate the process of information locating, information delivery infers the need for information by monitoring the low-level activities of users, and then autonomously locates and delivers information based on the inferred needs (Nardi, Miller et al., 1998). Information delivery is needed to take advantage of the large number of potentially useful components contained in the (L4 – L3) area of Figure 1 (Belkin, 2000). The fundamental challenge in making information delivery systems useful is to exploit the working context and the distinct information needs of each user to present only the information that is related to the task-at-hand and is not yet known to the individual user (Fischer

and Ye, 2001), rather than bombarding users with decontextualized and irrelevant information. A well-known example of a decontextualized information delivery system that is almost universally unused is Microsoft Office's *Tips of the Day*.

Information delivery explores the power of *implicit communication channels* (Fischer, 2001) that are established when reuse repository systems are integrated with development environments (Fischer, Nakakoji et al., 1998). Such integration creates a shared workspace between software developers and reuse repository systems. Through this shared workspace, reuse repository systems can infer the task of software developers by analyzing their partially written programs and can then deliver task-relevant components without explicit queries from software developers. Furthermore, reuse repository systems can create and maintain *user models* to represent particular software developers' existing knowledge of the reuse repository to ensure the delivery is personalized to varying individual needs.

Retrieval-by-Reformulation. Due to the aforementioned conceptual gap and users' unfamiliarity with the information space of HFAs, many users are unable to create a well-defined query on their first attempt to locate relevant information (Mili, Yacoub et al., 1999). Information systems can, at best, retrieve information that matches the queries submitted by a user, and the retrieved information may not necessarily match the user's real intentions, many of which are not articulated. *Retrieval-by-reformulation* (Williams, 1984) is the process that allows users to incrementally improve their queries after they have familiarized themselves with the information space by evaluating previous retrieval results. Retrieval-by-reformulation is especially important in information delivery systems in which information needs are inferred. By combining information delivery and retrieval-by-reformulation, the information location process becomes a collaborative one in which computers and users complement each other's strengths (Terveen, 1995).

2.3 Knowledge-Based Systems and Reuse

The influence of knowledge-based systems on reuse is twofold. First, reuse repository systems can act as software developers' assistants to supplement their insufficient knowledge about components. Second, knowledge-based approaches can be used to infer the needs for components from low-level user activities through the implicit communication channel.

Knowledge Augmentation. Theories about distributed cognition (Salomon, 1993) have revealed that a cognitive activity is primarily determined by its surrounding environment, which includes information present in both the workspace and the memory of human beings. Subsequent problem-solving actions are chosen by incorporating new information from the developer's memory triggered by cues present in the workspace (Simon, 1996). This explains why software developers with differing knowledge often choose very different approaches to develop the same task (Visser, 1990). For example, for the same task, a software developer who recalls a certain component that can be

reused in the task may take a bottom-up approach to design a program that is centered on the component, whereas another developer who does not know or recall that component may take a top-down approach to further decompose the task (Sen, 1997).

Through information delivery, unknown components can be reused in a manner similar to that of known components. Because timely delivered components based on cues in the workspace become a part of the immediately accessible information in the workspace, they can be regarded as the results of recall automated by computers, and motivate software developers to take a design approach that favors reuse. With the information delivery mechanism, all components in the reuse repository, whether known or not, may possibly actively contribute to the software development process.

Finding Task-Relevant Components with Similarity Analysis. The two basic approaches to inferring the high-level goals of users from their low-level activities and then finding task-relevant information to help them accomplish the task are *plan recognition* and *similarity analysis*. Due to the difficulty of recognizing plans from an unfinished program, we use the similarity analysis approach (Figure 2). Similarity analysis is based on the following assumption: If the current working situation, defined by the self-revealing information in the workspace, is similar enough to a previous situation in which information X was used, then it is highly possible that information X is also needed in the current situation.

Software developers often use meaningful comments and identifier names to communicate the concept or the functional purpose of a program (Soloway and Ehrlich, 1984; Anquetil and Lethbridge, 1998; Michail and Notkin, 1999); *doc comments* of Java are specifically introduced for that purpose. Other self-revealing information includes the signatures of modules that define the types of input and output data (Zaremski and Wing, 1995). Therefore, the relevance of a component to the task-at-hand can be determined by the *conceptual similarity* between the comments

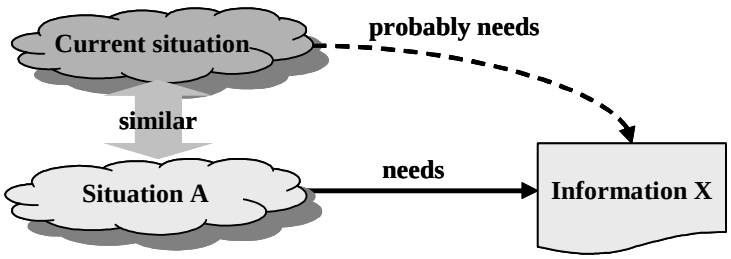


Figure 2: Similarity analysis

and identifiers in the program being developed and the textual documents of components in the repository, and the *signature compatibility* between the signatures of programs under development and those of components.

Latent Semantic Analysis. Latent semantic analysis (LSA) (Landauer and Dumais, 1997) is a free-text indexing and retrieval technique that takes semantics into consideration. It can be used to determine the conceptual similarity between the task-at-hand and components in the repository. From a large volume of training documents in a specific

domain, LSA first creates a domain-specific semantic space of words to capture the overall pattern of their associative relationship. Text documents and queries are represented as vectors in the semantic space, based on the words contained; and the similarity between a query and a document is determined by the distance of their respective vectors. The semantic space created by LSA is similar to the knowledge net that a human acquires about words through reading (Kintsch, 1998), and therefore has the potential to reduce the conceptual gap between situation model and system model in locating components.

3. CODEBROKER: DELIVERING TASK-RELEVANT AND PERSONALIZED COMPONENTS

Guided by our conceptual framework, we have designed, implemented, and evaluated a reuse-conducive development environment called *CodeBroker* (Figure 3), which encourages and enables software developers to reuse unknown components through the autonomous delivery of task-relevant and personalized components. It supports Java developers in reusing components *within* their development environment, *Emacs*, which is augmented by the *RCI-display* (Reusable Component Information display, the lower buffer in Figure 3), where task-relevant and personalized components are autonomously shown in response to the change of programming context in the editor.

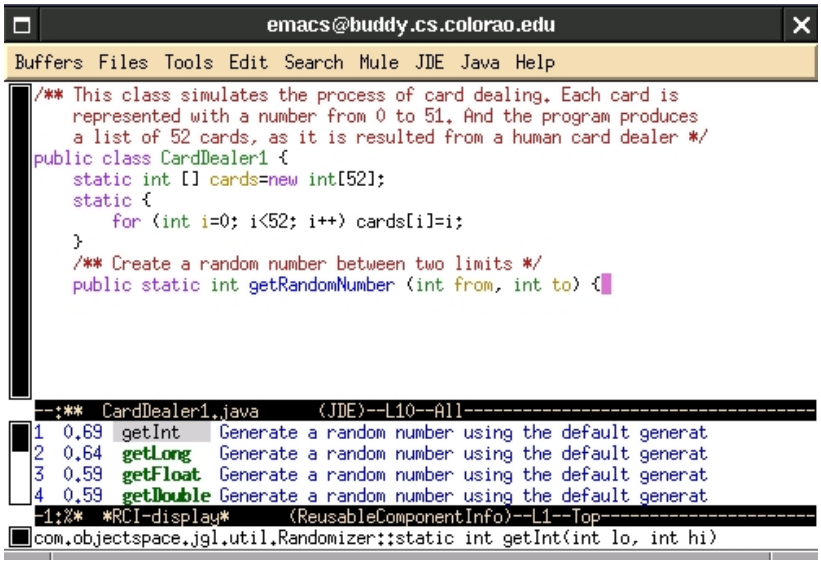


Figure 3: An example of the use of CodeBroker

This screen image shows what a developer using *CodeBroker* sees. The developer wants to write a method that creates a random number between two integers. Based on how the developer describes the task in the doc comment and the signature before the cursor, several components are delivered in the RCI-display (the lower buffer). The first of these delivered components, `getInt`, is a perfect match and can be reused immediately.

3.1 Overview of the CodeBroker System

CodeBroker consists of an *interface agent* and a *back-end search engine* (Figure 4). The interface agent runs continuously as a background process in *Emacs* and mediates the software developer's interaction with the back-end search engine. The back-end search engine acts like many other existing software reuse repository systems: It accepts queries and returns components that match the queries from the component repository. The component repository contains indexes created by *CodeBroker* from the standard Java documentation that *Javadoc* generates from Java source programs, and links to the Java documentation system.

In *CodeBroker*, however, a software developer does not need to directly interact with the search engine; the interaction with the reuse repository system is automated by the interface agent, which autonomously extracts reuse queries from the program editor and delivers personalized retrieval results. When a software developer enters a doc comment in the editor, the interface agent extracts the contents of the doc comment and creates a query. The query is

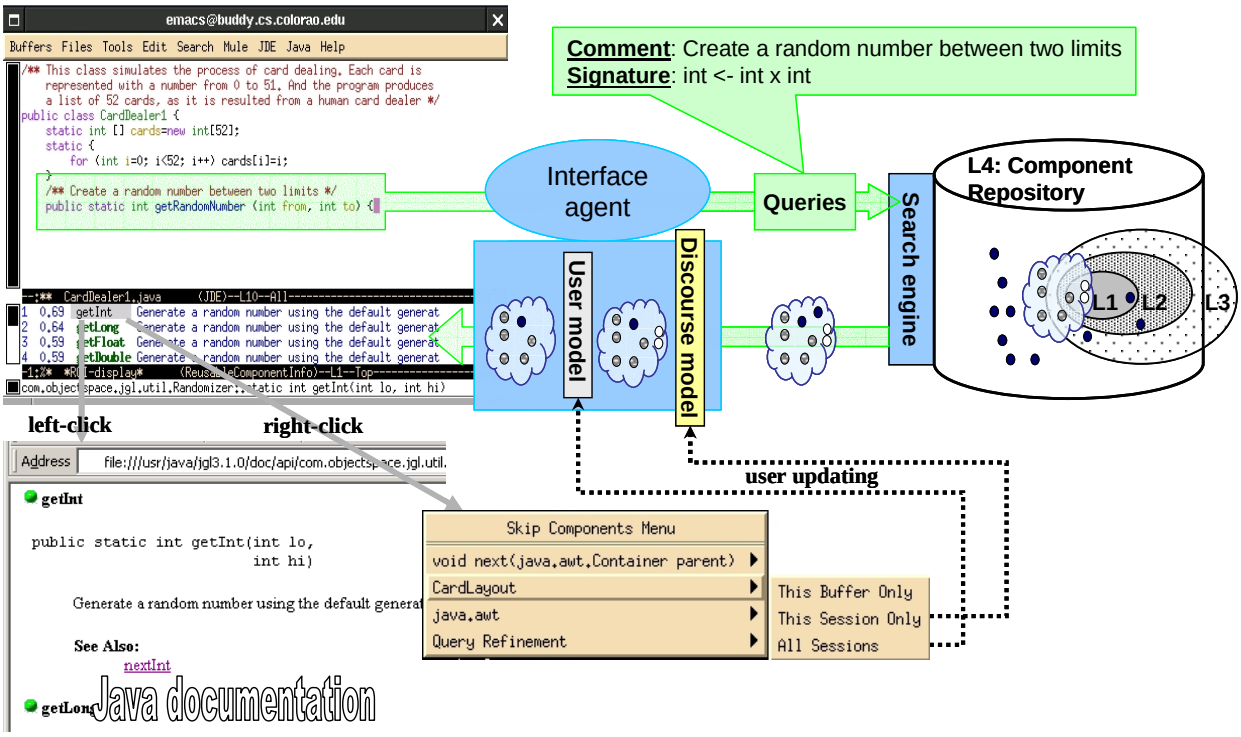


Figure 4: The system architecture of *CodeBroker*

Components that match the queries, which are extracted from doc comments and signatures, are automatically retrieved from the component repository. The retrieved components that are not included in the discourse model and the user model are delivered into the workspace. Discourse models (see Section 3.4) remove task-irrelevant components (black dots), and user models (see Section 3.5) remove known components (unshaded dots). Discourse models and user models can both be updated by users through the `Skip Components Menu`. Users who want to know more about a component can go to the Java documentation by clicking on the delivered component.

passed to the back-end search engine, which retrieves a list of components that matches the query by using LSA (see Section 3.2). The retrieval results are passed back to the interface agent. However, the interface agent does not deliver all the retrieval results to the developer; it personalizes the retrieval results by removing those components that are included in either the *discourse model*, which includes components that the developer, in previous interactions with the system, has indicated are of no interest (see Section 3.4), or the *user model*, which contains components already known to the software developer (see Section 3.5). The delivered components are shown in the *RCI-display* immediately after the doc comment is entered.

If the developer cannot instantly find what he or she wants from the delivery based on the doc comment, the developer can continue programming by defining the signature of the method. As soon as the signature definition is finished (the left bracket '`{`' before the cursor), the interface agent extracts the signature and combines it with the preceding doc comment to create a new reuse query. Upon receiving the reuse query, the back-end search engine retrieves a new list of matching components by combining LSA and signature matching (see Section 3.2). The retrieval results are again delivered by the interface agent into the *RCI-display* after filtering via the discourse model and the user model. For example, the first component in the *RCI-display* in Figure 3, which matches both the doc comment and the signature, does exactly what the developer wants and can be reused right away.

CodeBroker presents information on reusable components with three different layers of abstraction. The first layer is the *RCI-display*, in which 20 components (the number can be customized) are shown according to their task relevance, and each component is accompanied by its rank of relevance, relevance value, name, and synopsis. To reduce intrusiveness (Fischer, Nakakoji et al., 1998), users are not required to interact with the system if they are not interested in the delivered components. If users are interested in certain components in the *RCI-display*, they can trigger the presentation of the second layer of information with mouse movements. When the mouse cursor is moved over the component name, the signature of the component is shown in the mini-buffer (the last line of *Emacs* in Figure 3); and when the mouse cursor is over the synopsis, words contributing to the relevance between the component and the task-at-hand are shown in the mini-buffer to reveal why this component is retrieved and to help software developers refine their queries if necessary. The third layer of information, which is the most complete description of a component, is shown in an external HTML browser. A left-click on the component name brings up the full *Javadoc* documentation for the component (Figure 4).

If the software developer feels too many irrelevant components are delivered in the *RCI-display*, he or she can activate the `Skip Components Menu` by right-clicking on delivered components to filter them out (Figure 4). Filtering can be applied at three levels of granularity: (1) filtering out the component itself by choosing the first item in the menu, (2) filtering out all components from its class by choosing the second item, or (3) filtering out all components from its package by choosing the third item. Three commands exist for each chosen item. The first command, `This Buffer Only`, removes the chosen item from the *RCI-display* buffer; the second command, `This Session Only`, not only removes the chosen item from the buffer, but also adds it to the discourse model (see Section 3.4); and the third

command, `All Sessions`, both removes the chosen item from the buffer and adds it to the user model (see Section 3.5).

3.2 Locating Task-Relevant Components

CodeBroker explores both implicit and explicit communication channels to deliver task-relevant components. An *implicit communication channel* refers to the passage of information from a user to a computer system that is not explicitly initiated by the user but is implicitly inferred by the computer system, and an *explicit communication channel* refers to the passage of information that is explicitly initiated by the user (Fischer, 2001).

By embedding its interface agent into *Emacs*, the development environment, *CodeBroker* creates an *implicit communication channel* between software developers and reuse repository systems. Through the implicit communication channel, *CodeBroker* autonomously extracts reuse queries from partially written programs and delivers components that match the extracted queries; it then uses *explicit communication channels*—retrieval-by-reformulation (see Section 3.3) and discourse models (see Section 3.4)—to allow software developers to improve the task-relevance of delivered components explicitly and incrementally. This section describes the mechanisms of extracting queries and retrieving components; the following sections describe retrieval-by-reformulation (Section 3.3) and discourse models (Section 3.4).

Extracting Queries. Reuse queries are extracted from doc comments and signatures of the program on which a software developer is working. A software 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. This characterization is similar to the 3C model of Tracz (Tracz, 1990), who uses concept, content, and context to describe a component. Important concepts of a program are often contained in its informal information structure. Informal information includes structural indentation, comments, and identifier names (Soloway and Ehrlich, 1984), which are important beacons to understanding programs (Anquetil and Lethbridge, 1998; Michail and Notkin, 1999; Maletic and Marcus, 2001). One important constraint of a program is its type compatibility, which is manifested in its signature. For a reusable component to be easily integrated, its signature should be compatible with the environment into which it will be incorporated. Based on the assumption of *similarity analysis* (Figure 2), the code of a component is highly likely to be reused if it shows either *conceptual similarity*—the similarity between the textual document of a component and the doc comment extracted from *Emacs*—or *constraint compatibility*—the signature compatibility that exists between the signature of a component and the extracted signature, or both, to the programming task at hand.

Retrieving Similar Components. When a doc comment is entered by the user, *CodeBroker* retrieves from the repository the components that show conceptual similarity. When a signature is entered, *CodeBroker* retrieves those

components that show both conceptual similarity to the doc comment immediately before the signature and constraint compatibility.

CodeBroker uses LSA (Landauer and Dumais, 1997) to compute the conceptual similarity. LSA starts by creating a semantic space with a large corpus of training documents in a specific domain. We have used the Java Language Specification, Java API documents, and Linux manuals as training documents to acquire a level of knowledge similar to that of a Java programmer. The corpus contains 78,475 documents and 10,988 terms. A large term-by-document matrix is created in which entries are normalized scores of the term frequency in a given document. This term-by-document matrix is then decomposed, by means of singular value decomposition, into the product of three matrices: a left singular vector, a diagonal matrix of singular values, and a right singular vector. These matrices are then reduced to *k* dimensions by eliminating small singular values. The value of *k* often ranges from 40 to 400, but the optimum value of *k* remains an open question and needs to be empirically determined. *CodeBroker* sets *k* to 300. A new matrix, viewed as the *semantic space* of the domain, is constructed through the production of the three reduced matrices. In this new matrix, each row represents the position of each term in the semantic space. Terms are re-represented in the newly created semantic space. The reduction of singular values is important because it captures only the major, overall pattern of associative relationships among terms by ignoring the noises accompanying most automatic thesaurus construction based simply on co-occurrence statistics of terms.

After the semantic space is created, each reusable component is represented as a vector in the semantic space based on terms contained. The extracted query is represented in the same way. The conceptual similarity of a query and a reusable component is thus determined by the Euclidean distance of the two representative vectors. *CodeBroker* retrieves 20 components (the number can be easily customized by users) that have the highest similarity values, and the similarity value is shown in the second column in the *RCI-display* (Figure 3).

The constraint compatibility is determined by *signature matching*. Signature matching is the process of determining the compatibility of two components in terms of their signatures (Zaremski and Wing, 1995). It is an indexing and retrieval mechanism based on type constraints. The basic form of the signature of a method is:

```
Method:InTypeExp->OutTypeExp
```

where `InTypeExp` and `OutTypeExp` are type expressions resulting from the application of a Cartesian product constructor to all their parameter types. For example, for the method,

```
int getRandomNumber (int from, int to)
```

the signature is

```
getRandomNumber: int x int -> int
```

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 conformant if they are formed by applying the same type constructor to structurally conformant types.

This definition of signature matching is very restrictive because it misses some components whose signatures do not exactly match, but which are, in practice, similar enough to be reusable after slight modification or with wrappers added. *Partial signature matching* relaxes the definition of structural conformance of types: A type is considered as conforming to its more generalized form or more specialized form. For procedural types, if there is a path from T_1 to T_2 in the type lattice, T_1 is a generalized form of T_2, and T_2 is a specialized form of T_1. For example, in most programming languages, *integer* is a specialized form of *float*; and *float* is a generalized form of *integer*. For object-oriented types, if T_1 is a subclass of T_2, T_1 is a specialized form of T_2, and T_2 is a generalized form of T_1.

The constraint compatibility value between two signatures is the product of the conformance value between their types. The type conformance value is 1.0 if two types are in structural conformance according to the definition of the programming language. It drops a certain percentage if one type conversion is needed, or there is an immediate inheritance relationship between them, and so forth. The signature compatibility value is 1.0 if two signatures exactly match.

3.3 Supporting Retrieval-by-Reformulation

Doc comments and signatures may not describe the task-at-hand completely and precisely. Furthermore, current information retrieval algorithms, including LSA, are unable to retrieve all of the task-relevant information and only the task-relevant information (Salton and McGill, 1983). *CodeBroker* unavoidably delivers some irrelevant components and misses some relevant components. The *retrieval-by-reformulation* (Williams, 1984) interface (Figure 5) in *CodeBroker* enables software developers to incrementally reformulate reuse queries, after they have

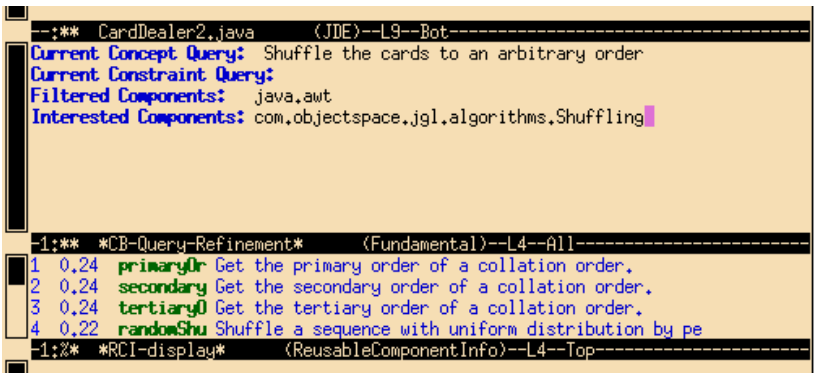


Figure 5: The retrieval-by-reformulation interface

studied the delivered components, until they are able to locate what they want.

The retrieval-by-reformulation interface is an *explicit communication channel*, which must be activated by software developers to refine queries and limit the range of retrieval. Most reuse repositories are organized hierarchically. For example, components in Java are placed in different packages and classes according to their application domains. Most development tasks involve only a part of the repository, and software developers are not interested in components from irrelevant packages and classes. Through the retrieval-by-reformulation interface, software developers can exclude components from certain packages and classes by adding the names of these components to the `Filtered Components` field, or limit the search to packages and classes of interest by adding package or class names to the `Interested Components` field (Figure 5).

Although the interface can also be used as a traditional search interface, software developers who do not know the structure of the repository well enough may not be able to specify the interested or uninterested parts at their first searching attempt. Autonomously delivered components can familiarize software developers with the repository and enable them over time to formulate reuse queries that are closer to the *system model* of the reuse repository (Kintsch, 1998).

3.4 Creating and Using Discourse Models

Doc comments and signatures describe the immediate programming task, namely, the module that the software developer is going to develop. A module is only a part of the whole development task, and the functionality of the module is deeply connected with other modules that have been developed so far. Therefore, software developers' interactions with the system in the development of previous modules provide a *discourse* to interpret the current development activity and to limit the applicability of information in the current situation. This is similar to the conversation structure in natural language, in which a new utterance is interpreted by the listener in light of the conversational discourse defined by previous interactions.

The interaction history between a software developer and *CodeBroker* in a development session is captured in a *discourse model*, which is used as a filter to improve the task-relevance of delivered components. A development session is defined by the software developer, who starts and ends the session by activating and deactivating the *CodeBroker* system, respectively.

Each development session starts with an empty discourse model that is incrementally updated by the software developer as he or she interacts with the system. Discourse models in *CodeBroker* contain components that do *not* interest software developers in the current development session because it is often much easier for a user to identify misfits than fits (Alexander, 1964). As a software developer starts a development session, he or she can gradually add components that are not of interest to the discourse model by using the `Skip Components Menu` (Figure 4), which tells the *CodeBroker* system not to deliver those components again in the same session. Figure 6 shows an

```
;; Discourse model for the session started 08:53:12, Nov. 2, 2000.
(("java.awt") ;; Package added at 08:55:53.
 ("java.text" ("ParsePosition")) ;; Class added at 08:56:30.
 ("java.lang" ("Runtime")) ;; Class added at 09:56:45.
)
```

Figure 6: An example discourse model

example of a discourse model in which the subject was asked to write a program that creates, in a specified directory, the backups of a set of files. This programming assignment, from one of our evaluation experiments, involved several tasks (i.e., writing several methods). The first task the subject conducted was to write a method to parse command lines. He wrote the following doc comment (Figure 7):

```
/** Parse the command line args and copy files */
```

When this doc comment was entered, *CodeBroker* delivered a list of task-relevant components (see the *RCI-display* in Figure 7). The subject noticed that the first component `setActionCommand` in the *RCI-display* was from the

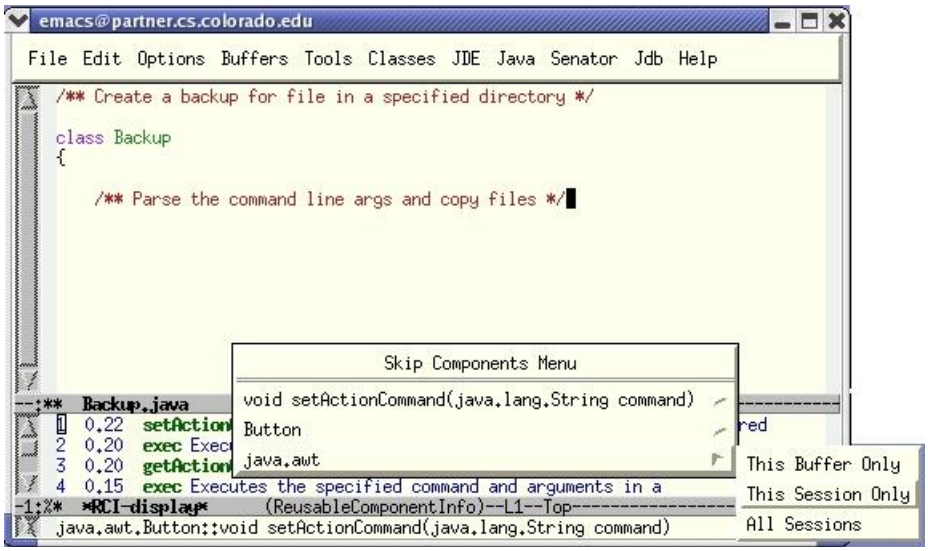


Figure 7: Updating the discourse model

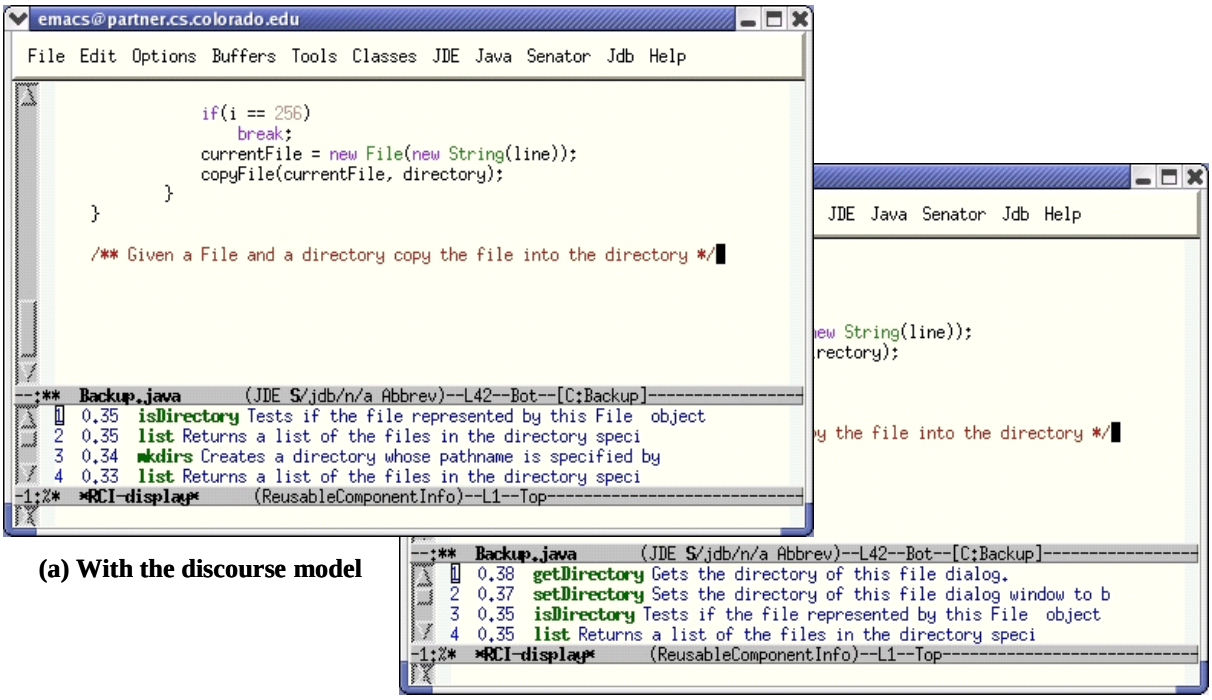
`java.awt` package. Because the subject was sure that he would not need any components from that package for his program, he added `java.awt` to the discourse model with the `Skip Components Menu` by choosing the `This Session Only` command.

After finishing the first task, the subject started to work on the second task: copying a file into a specified directory. After using as a filter the discourse model the subject had updated in his first task, *CodeBroker* delivered several components in response to the subject's following doc comment (Figure 8(a)):

```
/** Given a file and a directory copy the file into the directory */
```

He used the `isDirectory` (the first component in the *RCI-display*) component to test if the input directory name already existed as a directory, and `mkdirs` (the third one) to create a new directory if the directory did not yet exist.

However, if the subject had not added `java.awt` to the discourse model in his first task, the first two components delivered by *CodeBroker* (Figure 8(b)) would have been `getDirectory` and `setDirectory`, both of which were from the `java.awt` package and were not reusable in the subject's programming task; thus, the subject would not have been able to see the `mkdirs` component immediately.



(a) With the discourse model

(b) Without the discourse model

Figure 8: Deliveries with and without the discourse model

Both a discourse model and the `Filtered Components` field in the retrieval-by-reformulation interface (Figure 5) are used to remove irrelevant components specified by software developers. However, the former is used not only in the current delivery but also in all following deliveries in the same development session, whereas the latter is used only for the current delivery. Such a design is meant to give software developers different levels of control of the scope of component location according to their needs.

3.5 Delivering Personalized Components

Delivering a component that is well known to a software developer is not desirable. Because each software developer has unique knowledge about the reuse repository, *CodeBroker* needs to personalize its delivery to each developer's individual needs.

```
;; user model for Jeff
(
("java.applet"
    ("Applet"
        ("getParameterInfo")) ;; Added by Jeff at Thu 2 08:20:10 2000
)
("java.io"
    ("File"
        ("exists" "Thu Nov 2 08:35:49 2000" "Nov 2 08:15:10 2000" "Nov 2 08:
        ("isAbsolute" "Thu Nov 2 08:36:31 2000" "Nov 2 08:19:15 2000" "Nov 2
    ("CharArrayWriter"
        ("toCharArray")) ;; Added by Jeff at Thu 2 09:00:11 2000
)
("java.net") ;; Added by Jeff at Thu 2 09:15:11 2000
)
```

Figure 9: An example user model

A user model is a Lisp list with the following format:

```
(package
   (class
      (method use-time use-time use-time …)))
```

where the use-time field indicates when the developer reused the component. No use-time field means the component was added by the user. An empty class field or method field means the whole package or class is known to the developer.

CodeBroker uses *user models* (Figure 9) to represent software developers' knowledge about the reuse repository. User models contain both well-known and vaguely known components (L1 and L2, respectively, in Figure 1). The interface agent of the *CodeBroker* system removes only well-known components contained in the user model from the retrieval results returned by the search engine because, although software developers can retrieve L2 components by themselves, automatic delivery can save the locating time.

The contents of user models are collaboratively maintained by the system and users. *CodeBroker* creates the initial user model by analyzing the Java programs the software developer has created so far. A software developer can explicitly adapt his or her user models. When a known component is delivered and the user does not want the same component to be delivered again, he or she can use the `Skip Components Menu` (Figure 4) to add the component, its class, or its package to his or her model. Such user-added components do not have a use-time field in the user model; they belong to L1 in Figure 1.

CodeBroker implicitly updates user models when it observes that software developers invoke a method component during their programming. It uses the following heuristics to determine when a method component is invoked. A

method invocation in Java is followed by a left parenthesis. Whenever a left parenthesis is entered in the editor, after *CodeBroker* has excluded the non-method invocation cases, such as the Java `for` statement, it scans back to extract the name of the method. Because a method name may not be unique in Java, *CodeBroker* needs to determine its class and package to add it to the user model. If the method is an instance method, *CodeBroker* determines its class by looking up the declaration of the variable that precedes the method. If the method is a class method and its class is not included in the method invocation statement, *CodeBroker* looks up all imported classes of the program to find the class that has that method. If the class is not unique in the repository, *CodeBroker* picks the package that is imported in the beginning of the program with the Java `import` statement. Only method components are implicitly added to user models in *CodeBroker* because the software developer may not know the entire class, even if a method of the class is reused. The component added to user models by the system has a use time, which is the time the component is detected to be invoked in the editor. Components with more than three use times (the number is customizable) are considered well known (i.e., included in L1 in Figure 1); components with fewer than four use times are considered vaguely known (L2).

Both the user model and the discourse model are used as filters by *CodeBroker* to tailor the delivery to a particular context and a particular developer by removing unwanted components from retrieval results, and are maintained through the same interface (`Skip Components Menu`). However, they are conceptually different. The discourse model includes components that are not of interest in the *current* session no matter whether the software developer knows them or not; those components need to be delivered again when a different development session starts. Thus, the use of the discourse model results in the *context-awareness* of the *CodeBroker* system because, even for the same reuse query and the same developer, different components are delivered when the context (defined by the interaction history between the developer and the system in one development session) under which the query is extracted is different. User models include components known to individual software developers no matter whether these components are related to the current session or not, and they persist through different development sessions. Thus, the use of the user model results in the user-awareness of the *CodeBroker* system because, even for the same reuse query, different components are delivered for different users.

The discourse model and the user model also are implemented differently. The discourse model is stored in the internal memory and is re-initialized to empty when the system is re-started, whereas the user model is stored in permanent storage and is loaded into the system each time it 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.

4. EVALUATION

To assess the usability and usefulness of *CodeBroker*, we conducted empirical evaluation experiments with software developers. The reuse repository used in our evaluation experiments included 673 classes and 7,338 methods from

the Java 1.1.8 core library and the JGL 1.3 library (created by Objectspace, Inc.). Through the experiments, we attempt to answer the following questions:

- Does *CodeBroker* enable software developers to reuse unknown components?
- Does *CodeBroker* encourage software developers to explore the possibility of reuse?
- Is the technical approach taken by *CodeBroker*—inferring reuse queries from doc comments and signatures—good enough to find components relevant to the task-at-hand?
- Do discourse models improve the relevance of delivered components?
- Do user models contribute to the personalization of component deliveries?

4.1 Recall and Precision

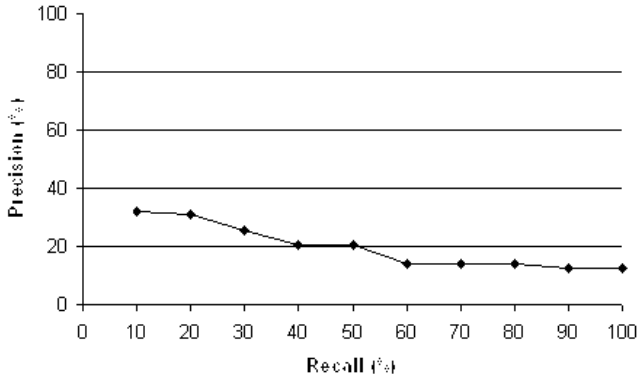


Figure 10: The recall-precision curve

Information retrieval systems are conventionally evaluated by *recall* and *precision* (Salton and McGill, 1983). *Recall* is the proportion of relevant material actually retrieved in answers to a search query; and *precision* is the proportion of retrieved material that is actually relevant. Figure 10 shows the recall-precision curve for the results of executing 19 queries in *CodeBroker* (see Table 1) for examples of queries and relevant components). One half of the queries were created by us, and the other half were collected from empirical experiments and frequently asked questions (FAQs) in Java-related newsgroups. The recall-precision data shown in Figure 10 are lower than those reported in the evaluations of other reuse repository systems (Frakes and Pole, 1994). However, retrieval systems are comparable only when all the queries and the criteria for relevance are the same. Our criteria for relevance were very strict because we considered as relevant only those components that could actually be reused in implementing the tasks described by the queries (see Table 1). Furthermore, Frakes and Pole's experiments were conducted to find single Unix commands for a given task from a set of 120 Unix commands, whereas our experiments were to find components that could be combined to implement a programming task from a repository that contained 7,338 items.

Further experiments are needed to compare the effectiveness of the retrieval mechanism used in the *CodeBroker* system with that of other systems by subjecting all systems to the same test conditions. Because the major goal of our research is not the invention of a new retrieval mechanism, but the new interaction style with a reuse repository system that is conducive to reuse, our future research includes identifying through experiments and incorporating into the *CodeBroker* system a better retrieval mechanism that is also based on the free-text information retrieval technique.

Table 1: Examples of queries and relevant components

Queries	Relevant Components
Determine if it is a leap year	`GregorianCalendar.isLeapYear`

Change the file name	`File.renameTo`
	`File.canRead`
	`File.canWrite`

Append two strings	`StringBuffer.append` `String.concat`
Given a file and a directory, copy the file into the directory	`File.getAbsolutePath` `File.getName`
Check if a directory exists; if not then create it	`File.mkdir` `File.isDirectory`

4.2 The Structure of the Experiments

Five subjects who had extensive software development experience voluntarily participated in the evaluation experiments. Their expertise in Java varied from medium to expert (Table 2). Our experiments adopted both the *multi-project variation* approach, in which one subject conducted two or three different projects, and the *replicated project* approach, in which one project is conducted by two or more subjects (Basili, Selby et al., 1986).

Twelve experiments were conducted. In each experiment, the subject was asked to implement a predetermined small task. Each task could be implemented with different combinations of components from the repository. The following is a sample task:

> Traditionally, Chinese write numbers with a comma inserted at each fourth number from the right. For example, 1,000,000 is written as 100,0000. Implement a program that transforms the Chinese writing format (100,0000) to the Western format (1,000,000).

Before the experiment, *CodeBroker* first created initial user models for the subjects by analyzing the Java programs they had developed recently. Not surprisingly, the user model for the subject who had programmed with Java for 7 years and was a well-recognized expert Java programmer had the largest number: It included 605 methods from 164 classes and 32 packages (Table 2). Still, this was less than 10% of the methods included in the repository of the experiment. Because all subjects were quite experienced software developers who knew Java syntax very well, their

Table 2: Programming knowledge and expertise of subjects

Subject	S1	S2	S3	S4	S5
Years of programming in general	3 or 4	5 or 6	8	10+	10+
Years of programming with Java	10 months	4	4	7	5
Self-evaluation of Java expertise (1: novice 10: expert)	4	7	7 or 8	10	7
Initial user models (package#, class#, method#)	5, 23, 55	9, 53, 140	10, 51, 160	32, 164, 605	8, 41, 124

difference in Java programming expertise came mainly from differences in their knowledge level of the library components.

Subjects were instructed to follow their normal practice during the experiments. They were encouraged to take advantage of the components delivered by *CodeBroker*, but they were not forced to do so. They could also use their normal ways of locating components with books or the Java documentation system. Subjects were asked to describe their implementation plans for the given task before they started programming. We asked them to think aloud during the experiments, and we videotaped all experiments. Analyses were based on automatically logged data, transcribed videotapes, and post-experiment interviews in which we asked questions regarding their experience with *CodeBroker*.

4.3 Findings of Experiments

Table 3 shows the overall results of the experiments. Subjects reused delivered components during 10 of the 12 experiments. Column 3 shows the total numbers of distinct components reused in each experiment, which included the components delivered by *CodeBroker* as well as those the subjects either directly reused through their own knowledge or located through browsing or searching without *CodeBroker*. Column 4 shows the numbers of the components that the subjects reused from the deliveries made by *CodeBroker*. The 12 programs created by the subjects used 57 distinct components, 20 of which were delivered by *CodeBroker*.

Reusing Unanticipated Components. Of the 20 reused components that were delivered, the subjects did not anticipate the existence of 9 (see 5th column in Table 3). In other words, those 9 components could not have been reused without the support of *CodeBroker*, and the subjects instead would have created their own solutions. As two subjects commented in the interviews:

> "I would have never looked up the `roll` function by myself; I would have done a lot of stuff by hand. Just because it showed up in the list, I saw the `Calendar` provided the roll feature that allowed me to do the task."

> "I did not know the `isDigit` thing. I would have wasted time to design that thing."

Table 3: Overall results of empirical evaluations

1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*
Subject	Experiment no.	Total no. of distinct components reused	No. of distinct components reused from deliveries	Breakdown of reused components from deliveries		No. of reused components triggered by deliveries	Total no. of retrieved components	No. of components removed by discourse models	No. of components removed by user models	Rating on usefulness of the system (1: worst 10: best)
				No. of components whose existence was unanticipated (L4—L3)	No. of unknown components whose existence was anticipated (L2 & L3)					
S1	1	10	4	2	2	0	168	45	15	**7**
	2	3	1	1	0	1	28	10	0	
S2	3	7	1	1	0	0	140	0	5	**4**
	4	4	1	1	0	0	52	0	0	
	5	5	3	0	3	1	160	0	14	
S3	6	5	2	1	1	1	60	0	0	**8.5**
	7	4	3	1	2	1	20	0	1	
	8	3	0	0	0	0	60	0	0	
S4	9	4	3	0	3	0	80	7	0	**7**
	10	3	1	1	0	2	140	68	0	
S5	11	4	1	1	0	2	100	0	1	**8**
	12	5	0	0	0	0	420	0	0	
Sum		**57**	**20**	**9**	**11**	**8**	**1428**	**130**	**36**	

Reducing Locating Time. Although the subjects anticipated the existence of the other 11 components (see 6th column in Table 1), they had known neither the names nor the functionality, and had never reused them before. They might have reused the 11 components if they could manage to locate them by themselves. In interviews, subjects acknowledged that *CodeBroker* made locating them much easier and faster.

> "It beats browsing. Because the way that I normally would have done the task, I would do a lot of browsing and then write the code alongside. So this reduced the browsing and searching."

> "I did not have to start browsing and go through the packages, and I did not have to go through the index of methods. I could just go to the short list [*RCI-display*], find it and click it."

> "The key benefit of this [*CodeBroker*] is that it gives you methods for every class, not like this one [the Java documentation system] that you have to first find which class it is in and then go to the class. Although it has an index of methods, it is hard to find here [the Java documentation system]."

Snowball Effects of Deliveries. *CodeBroker* not only supported subjects in reusing components right off the deliveries, but also triggered them to reuse other unknown components (column 7) that were not directly delivered but were needed to reuse the delivered components. The reuse of one component often requires the reuse of other

supplementary components that are coupled through parameter passing or accessing common class variables. In the experiments, when those supplementary components were not known, subjects used the *CodeBroker* deliveries as starting points and followed the hyperlinks of the documentation system to learn and reuse them. Subjects had not known those triggered components before; the deliveries motivated software developers to reuse them.

Knowledge Augmentation. Information delivery not only encourages software developers to reuse components but also augments their abilities in constructing implementations centered on the delivered components that they have not known before. This observation was best illustrated with the different approaches taken by subjects S2, S3, and S5 when they implemented the sample task described in Section 4.2.

In describing his implementation plan, subject S3 anticipated that some methods from the `java.text.NumberFormat` class might help him read numbers in Chinese format and write it out in Western format, although he did not know exactly what those methods were nor what their functionality was. As a result, he successfully constructed his program concisely by using methods that were located by *CodeBroker* after he had limited the search to the `java.text` package with the retrieval-by-reformulation interface (Figure 5). Subject S5, who did not even know the existence of the `java.text` package, described his implementation plan as "to parse the number, take out the commas and insert the commas." As subject S5 started programming, he noticed a delivered component from the `java.text.NumberFormat` class, changed his original plan, and came up with a program similar to that of subject S3. Subject S2, who also did not know the `java.text.NumberFormat` class, described a plan similar to subject S5's original one. Because no component from the `java.text.NumberFormat` class was delivered based on his comments, he stuck to his original plan and constructed a different program from scratch.

In the experiments, we observed several other situations similar to the above example in which delivered components stimulated subjects to change their original plans to a new implementation approach that reused the delivered components.

Effectiveness of Delivering Components Based on Inferred Reuse Queries. *CodeBroker* infers reuse queries from doc comments and signatures contained in the program being worked on in the editor. The effectiveness of delivering task-relevant components depends on the quality of doc comments written by software developers and the retrieval mechanisms used.

The more knowledge subjects had about the repository, the more suited their doc comments were for retrieving relevant components. One subject described why he wrote one particular comment:

> "I knew there should be a class called `NumberFormat` or `DecimalFormat` having the method `format`...That's why I wrote the word 'format,' because I knew it would catch those."

As a result, he found what he expected from the deliveries of *CodeBroker*.

Different subjects had different styles of writing comments. Some wrote very long and elaborate comments to describe everything they wanted to do. Others wrote concise comments focusing on the major task of the program. Because descriptions of components in the repository were short and concise, the short and focused comments made the delivered components more task-relevant.

The ratio of the number of reused components to the total number of retrieved components is rather low (column 8 in Table 3 shows the total number of components retrieved in each experiment). This low ratio is due to the following reasons:

(1) Many components were retrieved while some subjects were to trying to locate some components that actually did not exist in the repository but somehow they believed should exist—to (note in Figure 1 that a part of L3 is outside of L4). For example, in experiment 12, the subject (S5) tried to find some components that could process events based on their priorities. His repeated search efforts by changing doc comments in the editor made the system retrieve 420 components, but none of these components were reusable because the components he was looking for simply did not exist in the repository. Similar things happened in experiments 5 and 10 to subjects S3 and S4, respectively. On the one hand, repeated failures in component searching are not desirable because they waste software developers' time; on the other hand, they indicate that *CodeBroker* is meeting one of its major design goals—to motivate software developers to attempt to reuse. In explaining why he repeatedly searched for the nonexistent components, subject S5 commented: "Having this system [*CodeBroker*], I would try to explore more. I would spend more time to see whether this thing exists or not."

(2) Of the retrieved components, some that were not reused by subjects were relevant. Many programming tasks can be implemented with different sets of components, and software developers only need a small subset of the retrieved relevant components to accomplish their tasks. A relatively objective, although not accurate, way of evaluating the effectiveness of retrieval mechanisms is to compute their recall and precision (Figure 10).

(3) The *CodeBroker* system retrieves and delivers components both when software developers enter a doc comment and when software developers enter the signature of a method. As will be discussed in the following paragraph, components retrieved and delivered at the entry of signature declaration were not reused at all. That means that about 50% of the retrieved components were of no use, but they did not substantially affect the effective use of the system or the retrieval performance of the system because those deliveries were hardly noticed by the subjects.

The signature-matching mechanism did not play too much of a role in the experiments. Only one subject tried once to look at the change of delivery when he finished the signature declaration of a method, but the system failed to improve the task-relevance of the delivery because no component in the repository was both similar in comments and

compatible in signatures to the task of the subject. In all other experiments, subjects shifted their attention to the delivery buffer immediately after they had written the comments. When they found the desired components, they moved back to programming and did not pay any attention to the delivery buffer until they wrote the next doc comment. The original design goal of adopting the signature matching mechanism in *CodeBroker* was to help developers find components that could be reused to replace the module under development. However, in the experiments, all subjects used the system to look for components that could be reused as parts of the module implementation instead of components to replace their intended implementation. The system is apparently more effective in delivering implementation parts than delivering replacement components.

Roles of Discourse Models. Discourse models, when created, improved the task-relevance of delivered components by filtering out components of packages and classes in which the subject was not currently interested. In four experiments, subjects added uninterested packages and classes to their discourse models, which removed about 10% of retrieved components from the deliveries (column 9). A careful examination of those removed components found that they could not be reused in implementing the tasks in the corresponding experiments. Because the discourse model depends on the interaction history between a software developer and the system in a particular development session that consists of several related tasks, it is expected to become more useful in natural settings than in the experiments in which subjects implemented only two or three unrelated tasks in a short time span. Further investigations are needed to confirm this hypothesis.

Roles of User Models. The experiments did not yield strong and conclusive data regarding the role of user models. Only 2% of the retrieved components were filtered by user models (column 10). That might be due to two reasons: (1) initial user models were not complete because subjects did not give us all the Java programs written by them; and (2) to observe the effectiveness of delivering unknown components, subjects were assigned tasks that involved the part of the repository they did not know very well, and, consequently, most delivered components were unknown. In the interviews, all subjects said they found that not too many known components were delivered. Nevertheless, user models helped and were needed to reduce the number of irrelevant components to be delivered because a careful examination of components removed by user models showed they could not be reused in the tasks. Similar to discourse models, the real value of user models is expected to be more evident when the system is used by software developers for a relatively long time.

The experiments reveal two design problems with the current user modeling approach in *CodeBroker*. The first problem is that because user models are kept permanently by *CodeBroker*, some subjects were concerned that if they added known components, the system would never deliver those components again. An interface for software developers to edit their user models might be able to alleviate this concern. The second problem was pointed out by a subject who said: “When you have programmed for a very long time, you may forget what you have used in your first program.” Counting the number of uses in the current user modeling approach is too simplistic; there should be

a forgetting mechanism incorporated to decide when to remove from user models those components that have not been reused by the software developers for a designated period of time. Another possible solution is to design a better presentation interface in the *RCI-display* window to allow software developers to turn on and off the user model-based filtering, or to arrange the order of delivered components so that known components may still be presented by the system as a reminder but will not hinder software developers from discovering unknown components.

Summary. Overall, the experiments have shown that information delivery can promote reuse by supporting the reuse of unanticipated components, reducing the cost of locating components, and augmenting software developers' capabilities in constructing new programs with components. Most subjects appreciated the support provided by *CodeBroker* and gave high ratings in terms of its usefulness, as shown in column 11 in Table 3, on a scale from 1 (totally useless) to 10 (extremely useful), and claimed that they would like to use *CodeBroker* as their daily programming environment. Even subject S2, who gave the lowest score (4), said, "It is right on the threshold that maybe I would use it."

5. DISCUSSION

The success of an information delivery system hinges on how many cues it can obtain from users' working environments to infer their needs for new information and retrieve that information (Nardi, Miller et al., 1998). Currently, the performance of *CodeBroker* is affected by the quality of doc comments and documents of components. Although LSA can reduce the conceptual gap between situation model and system model with fine-tuned domain-specific semantic spaces, the results are still far from satisfying, as we can see from the recall-precision curve (Figure 10). We are investigating more sophisticated mechanisms to retrieve and deliver components based on other cues in software development environments. For example, a software developer may write a program based on a known *design pattern or framework* (Gamma, Johnson et al., 1994), which places extra constraints on the type of components that can be reused. Such constraints can be utilized to improve the task-relevance of delivered components.

Reuse takes place in different phases of software development. The granularity of reusable components varies in different phases, but in all phases, software developers must be able to locate the needed components. *CodeBroker* is a "proof-of-concept" system that investigates the effectiveness of component delivery at the implementation level. This is important because it enhances the productivity of programmers. The opportunity of reuse depends on what software developers know of the repository when they are designing or implementing software. Delivering task-relevant and personalized reuse information can increase the reuse opportunity limited by the knowledge of software developers. The underlying design principles of *CodeBroker* can be extended to other phases of software development, and similar support can be provided. Software development is a knowledge-intensive activity, and

reusable components are only a portion of the knowledge needed. The information delivery mechanism is applicable not only to software components but also to other types of software development knowledge and in other phases of the software engineering process.

We are careful in extrapolating our findings from the experiments with *CodeBroker*, in which the repository consisted of components that were of very high quality, carefully documented, and highly trusted by software developers. Subjects were highly motivated to learn how to reuse those relevant components delivered by the system. We need to do more experiments to investigate whether the same conclusion holds with repositories that come from less respected sources. To answer this question, we will investigate the social aspects of software reuse, such as what makes software developers trust a component. The invisibility of software systems makes it impossible for software developers to judge the quality of components by their external appearances, and the complexity of software systems makes it difficult to judge the quality of components by their internal structure (Brooks, 1995). Our approach to addressing the trust issue is to look at the social context of components (Brown and Duguid, 2000): who produces them, who has reused them, and what the people who have reused them say about them. We are currently developing a *trust model* that can provide circumstantial evidence of the quality of a component based on the trust relationship among members of the developer community (Fischer, Scharff et al., 2003).

6. EVOLUTIONARY CONSTRUCTION OF REUSE REPOSITORIES

As mentioned in the introduction, the success of systematic reuse involves two intertwined issues: the creation and evolution of a reuse repository and the reuse of components from the repository. Accordingly, reuse-conducive development environments need to address the two issues simultaneously. The *CodeBroker* system is not yet a complete reuse-conducive development environment because it currently addresses only the latter issue: to motivate software developers to reuse by providing a better interaction interface to reuse repository systems. Our ongoing research efforts in extending *CodeBroker* are focusing on the former issue: to provide mechanisms that enable and encourage software developers to participate in the creation and evolution of the reuse repository.

Most of the past reuse research is conducted under the assumption that the creation and evolution of components and the reuse of components are two distinct phases because it is believed that a high-quality component repository can be produced and maintained by only a few select component developers (Poulin, 1999). However, the unexpected huge success of open-source software systems (Raymond and Young, 2001) prompts us to revisit this assumption. In our research toward the creation of reuse-conducive development environments, we consider the creation and evolution of components and the reuse of components as mutually enabling processes, both performed by the users of the reuse repository. This section outlines the theoretical differences between our approach, which we call the *decentralized, evolutionary paradigm*, and the traditional approach, which we call the *centralized paradigm*.

6.1 The Centralized Paradigm

The dominant paradigm of instituting a systematic reuse program in a software development organization is the centralized top-down approach (Prieto-Diaz, 1996). This paradigm makes a clear distinction between *producers* and *consumers* of reusable components (Fafchamps, 1994): A selected few component producers are dedicated to the development and maintenance of reusable components, which are then reused by application developers who are component consumers only. Such a clear distinction between component producers and consumers creates a high threshold for introducing reuse into the practices of software development organizations for the following reasons (Fischer, 2002):

(1) *Application domains are not static*; they change as quickly as business environments and practices, user requirements, and technology change. It is almost impossible to conduct a complete domain analysis and create a reuse repository that would be applicable in unforeseeable future applications.

(2) A dedicated component development team demands *huge initial investments* whose payoffs cannot be easily estimated; therefore, persuading top-level managers to commit to supporting reuse is an extra challenge.

(3) The *clear separation between component producers and consumers* creates an interest conflict between the two groups, with the former aiming at getting their produced components reused and the latter aiming at getting their work done with or without reuse. Application developers (component consumers) may view the reuse repository created by a separate component development team as an alien artifact that is forced upon them, and may thus develop the "Not Invented Here" syndrome to refuse reuse (Joos, 1994).

6.2 The Decentralized, Evolutionary Paradigm

For a long time, it was thought that such complex systems as operating systems and reuse repositories could be developed only in a centralized approach, or a cathedral style (Raymond and Young, 2001), as described in Section 6.1, to guarantee the high quality of the systems. The great success of such open-source software systems as Linux proves that complex systems can also be created incrementally in a bazaar style (Raymond and Young, 2001) through small contributions of large, and often distributed, user communities. The *Seeding, Evolutionary Growth, and Reseeding (SER)* model (Figure 11) that we have developed over the years for understanding the process of evolving complex systems provides a conceptual framework to understand this bazaar style (Fischer, 1998). We are instantiating the SER model in software reuse to find an approach for the evolutionary construction of reuse repositories by a large number of application developers rather than a few selected component developers.

In the *seeding* phase, component developers create an initial reuse repository that is intended to evolve by using one of the following two approaches: (1) developing components for a particular domain as the result of domain analysis

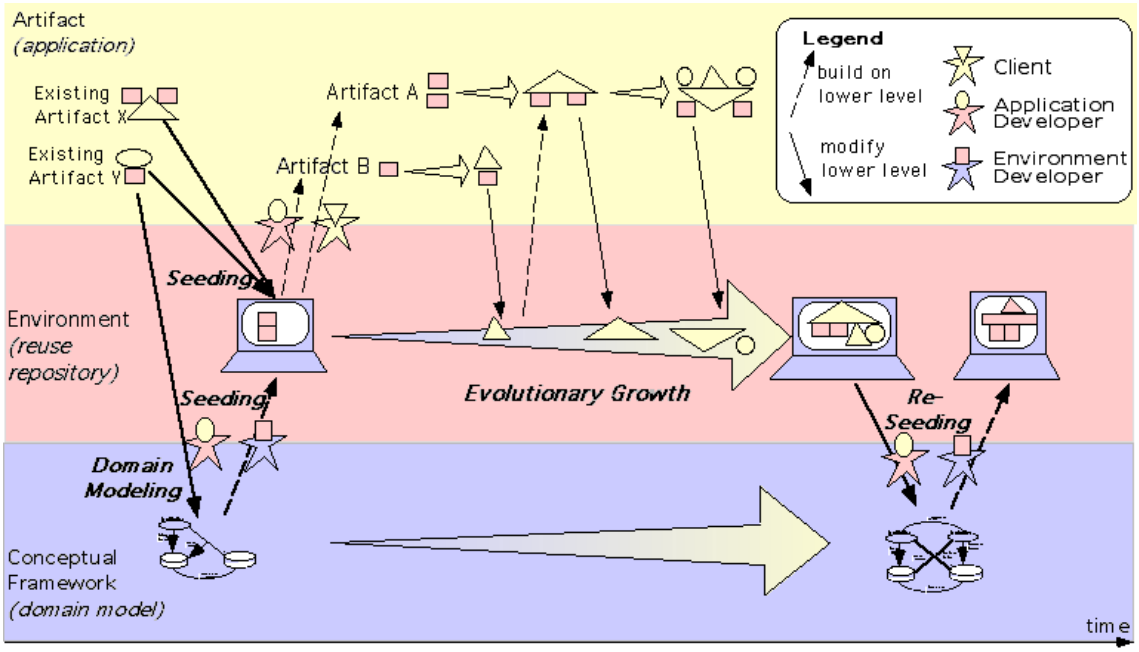


Figure 11: The Seeding, Evolutionary Growth, and Reseeding (SER) model

or product-line analysis (Griss, 2000); or (2) extracting reusable components directly from existing software systems (Etzkorn and Davis, 1997). The concept of a seed is based on the observation that an initial repository does not need to be perfect because an 80% solution that can be deployed and evolved incrementally is preferable to waiting for a 100% solution that never happens (Schmidt, 1999).

The *evolutionary growth* phase is one of unplanned evolution as the seed is reused by application developers in their work. During this phase, the seed plays two roles: It provides resources for work and it accumulates the products of work. Evolutionary growth happens as application developers modify components in the reuse repository for their own purposes and then contribute the modified components back to the reuse repository. During this evolutionary growth phase, bug-fixes, better documentation, the generalization or specialization of original components, and new components can all be captured and added to the reuse repository.

A *reseeding* process is needed when the growth makes the repository too chaotic to grow further. During reseeding, the repository is reorganized, and its components are refactored and generalized, based on their use and the information added by software developers during the evolutionary growth. In this phase, components that have not been reused over a long period of time would be removed.

Without application developers who are motivated to contribute, reuse repositories cannot evolve. Factors that affect motivation are both cognitive (intrinsic) and social (extrinsic). The precondition for motivating software developers to contribute is that they must derive an intrinsic satisfaction in accomplishing their tasks and goals by benefiting from the reuse repository at first; then they will reciprocate with their own work for the benefit of others (Grudin,

1994; Karat, Karat et al., 2000). The *CodeBroker* system increases the opportunity for application developers to obtain immediate benefits from the existing reuse repository through its delivery mechanism. To abide by the social norm of generalized reciprocity, software developers are more likely to feel obliged to return the favor by reciprocating with their own work for the benefit of others (Nahapiet and Ghoshal, 1998). Intrinsic motivation is positively reinforced when social conventions of the community recognize and reward such behaviors (Ye and Kishida, 2003). In addition to reducing the technical and cognitive difficulties in reusing components and contributing to the reuse repository, reuse-conducive development environments need to enable developer communities to form, develop, and maintain a sense of shared identity around reuse repositories. Studies of successful virtual collaborative communities, such as open-source communities and the expert-exchange website (http://www.expert-exchange.com/), have revealed that explicit recognition of contributing members, reputation enhancement, and positive peer pressure are effective in motivating users to become active contributors in communities (Fischer, Scharff et al., 2003). These insights will be incorporated in a future version of *CodeBroker* to address the social issues of software reuse.

As many studies and experiments have shown, individual differences in motivation exist: Some people are more motivated than others to contribute (Revelle, 1993). The technological difficulties in contributing might thwart those less motivated. We are currently extending *CodeBroker* to support the easy contribution of modified components to the reuse repository. Our goal is not to require that all application developers become active contributors to the reuse repository, but to provide technical means and social rewards to those application developers who are technically capable and willing to contribute to the reuse repository (Fischer, 2002).

Enabling application developers to participate easily in the evolution of reuse repositories gives them the sense of ownership of the repositories and can effectively overcome the "Not-Invented-Here" syndrome because the reuse repositories are owned not by the component developers, but by the application developers themselves. Having participated in the evolution of a reuse repository, application developers would likely consider the reuse repository as personally meaningful and its utilization as important because, as observed by Rittel, "people are more likely to like a solution if they have been involved in its generation; even though it might not make sense otherwise" (Rittel, 1984).

6.3 A Comparison of the Two Paradigms

Reuse repositories are a subset of information repositories that include knowledge management systems, digital libraries, design rationale systems, organization memory systems, and many others. The common problem faced by all of these repositories involves how to put information into the repository and how to extract useful information out of it. Our conceptualization of reuse-conducive environments is grounded in the general framework illustrated in Figure 12:

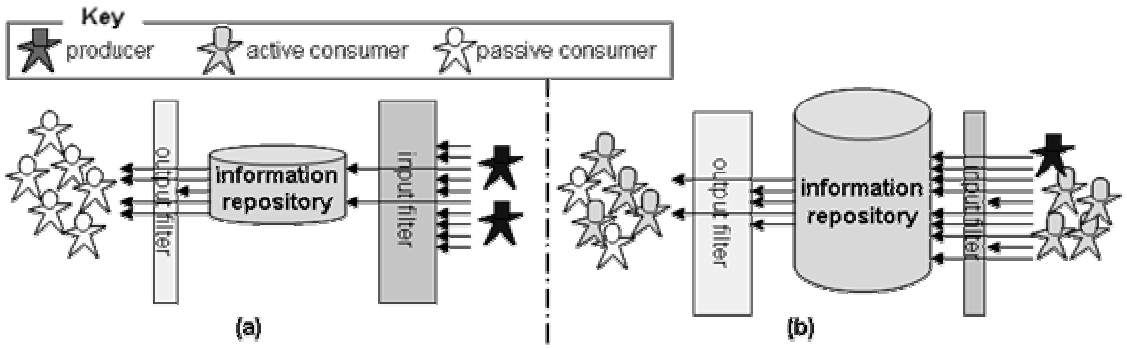


Figure 12: Two general paradigms of creating and using an information repository: (a) centralized, and (b) decentralized.

- The centralized paradigm (illustrated by Figure 12(a)) requires a thick, good input filter, which can be applied to select important and reliable information or to select a few dedicated information producers of high caliber, resulting in a relatively small information repository that contains only information of good quality but often misses other potentially useful information. It is relatively easy for information consumers, who are mostly passive, to locate and choose what they need from such an information repository. In addition to the problems described in Section 6.1, the major shortcomings of this paradigm are that potentially useful information might be left out and the growth of the information repository is limited.
- The decentralized and evolutionary paradigm (illustrated by Figure 12(b)) describes the collaborative construction of information repositories. It has a thin input filter that allows not only dedicated producers but also active consumers (or local developers (Nardi, 1993)) who are able and willing to contribute to put information into the information repository, resulting in a large information repository. This model requires a good, thick output filter, such as *CodeBroker*, that can provide information contextualized to the task-at-hand and the background knowledge of individual users.

7. RELATED WORK

This research has been heavily influenced by research efforts on both information delivery systems and reuse repository systems.

7.1 Information Delivery Systems

The simplest implementation of the information delivery mechanism is to deliver a piece of information without considering the working context, such as Microsoft Office's *Tips of the Day* and a similar research prototype, the

DYK (Did You Know) system (Owen, 1986). Most users find these systems at best of little help and at worst annoying, and they will turn them off if they know how.

Several of our own research systems have tried to increase the relevance of delivered information. ACTIVIST (Fischer, Lemke et al., 1985), an active help system for a text editor, uses a plan library to infer user goals from observed actions by matching them against the condition part of plans and suggests more efficient solutions to accomplish the same goals. *CodeBroker* and ACTIVIST are both totally embedded in the working environment, and in both, task-relevant information is delivered into the workspace. However, ACTIVIST delivers feedback information after users have finished their tasks, and the delivery is meant to improve their future work. Information delivered by *CodeBroker* is meant to influence the current task under execution. LispCritic (Fischer, 1987b) is another information delivery system that helps programmers to improve their programs and their programming skills. It uses program transformation rules to suggest a syntactical equivalent that is either a more cognitively efficient or computationally efficient solution after it has recognized a questionable code segment. Unlike *CodeBroker*, which makes use of both the semantic and syntactical information of programs, LispCritic has no knowledge about semantics.

Remembrance Agent (RA) (Rhodes and Maes, 2000) tries to augment human memory by displaying relevant documents. Like *CodeBroker*, RA also listens to a text editor and autonomously formulates a query based on the user's current focus. A back-end search engine is invoked to find relevant old emails and notes in the user's individual information space. RA deals with unstructured texts only, whereas *CodeBroker* relies on the semiformal structure of the program to extract needed information. In addition, *CodeBroker* also makes use of syntactical information. One shortcoming of RA is that it treats all documents the same, although its goal is to remind users of forgotten documents.

Letizia (Lieberman, 1997) assists users in browsing the web by suggesting web pages within a few links from the current page. Like *CodeBroker*, it aims at eliminating the context switch from a browsing interface to a search interface to streamline the exploration of web information. Web pages in a user's bookmark list are analyzed by using information retrieval techniques to create an interest profile. Suggestions are based on the similarity between web pages and the interest profile. Like *CodeBroker*, the suggestions made by the system are not meant to be the exact information needed by the user. They are the results of information reconnaissance (Lieberman, Fry et al., 2001) that surveys unknown information territory before the users are committed to entering it.

Information delivery has been explored in several other research prototypes of software development environments. Drummond and colleagues (Drummond, Ionescu et al., 2000) added to browsing systems an agent that infers the search goal of software developers by observing their browsing actions and delivers components that closely match the inferred goal. In addition, the *Argo* design environment (Robbins and Redmiles, 1998) is equipped with

computer critics that deliver general software design knowledge for software developers to reflect upon their current design.

7.2 Reuse Repository Systems

Research on reusable component repository systems is abundant. These systems differ from each other mainly in the component storage and retrieval mechanisms they adopt. A. Mili, R. Mili, and Mittermeir (1998) have conducted a comprehensive analysis of existing component storage and retrieval mechanisms along nine dimensions: nature of asset, scope of repository, query representation, asset representation, storage structure, navigation scheme, retrieval goal, relevance criterion, and matching criterion. Based on this analysis, the authors proposed to classify existing component repository systems into six broad families: information retrieval methods, descriptive methods, operational semantics methods, denotational semantics methods, topological methods, and structure methods. According to this classification, the similarity-analysis mechanism that *CodeBroker* uses to identify relevant components falls into the category of topological methods because it measures the similarity between the requirements for reusable components and the components in the repository by computing and combining both the conceptual similarity and the constraint similarity. *CodeBroker* also falls into the category of information retrieval methods because it attempts to retrieve relevant components by means of LSA when only conceptual queries extracted from doc comments are available.

CodeBroker differs from existing software reuse repository systems, however, in its attempt to extract reuse queries autonomously from the development environment. Most current reuse repository systems require that reusing software developers explicitly create reuse queries to represent what they actually want in order to find the components that are potentially reusable (Mili, Mili et al., 1998), and the systems then automate the process of finding the components that match the reuse queries formulated by the software developers. The ease of formulating such reuse queries is an important factor in determining the usability of reuse repository systems and hence their potential adoption by software developers (Mili, Mili et al., 1997). One of the major contributions of *CodeBroker* is to demonstrate the possibility of finding potentially reusable components without being given explicit reuse queries. *CodeBroker* autonomously infers and creates reuse queries by analyzing and extracting the three aspects (concept, constraint, and code) of the program under current development. The retrieval mechanism (LSA and signature matching) that *CodeBroker* uses can be replaced by many other existing mechanisms whose query representations can be similarly inferred from the program under development. According to the aspect of the program on which the abstract representations of components and queries are based, we divide reuse repository systems into three categories: concept-based, constraint-based, and code-based.

Concept-Based Reuse Repository Systems. Most reuse repository systems that index and retrieve components based on concepts use free-text indexing. GURU (Maarek, Berry et al., 1991) indexes components based on their

textual documentation. Etzkorn and Davis have tried to use header comments (similar to the doc comments in *CodeBroker*) to index legacy object-oriented programs (Etzkorn and Davis, 1997). Comments and identifier names are also used for indexing in the system developed by DiFelice and Fonzi (DiFelice and Fonzi, 1998). Michail and Notkin have demonstrated the possibility of using identifier names only to find similar reusable components for comparison (Michail and Notkin, 1999).

Free-text indexing is easy both for those setting up a component repository and for programmers formulating reuse queries. Empirical studies have found that free-text indexing-based reuse systems, despite their simplicity, perform no worse, in terms of retrieval effectiveness, than other more delicate, effort-consuming repository systems (Frakes and Pole, 1994). Nevertheless, free-text indexing-based reuse systems do not directly support shortening the conceptual gap in query formulation.

One attempt to bridge this conceptual gap is to use structured representations and knowledge bases. Both CodeFinder (Fischer, Henninger et al., 1991) and LaSSIE (Devanbu, Brachman et al., 1991) use frames to represent reusable components. Frames in CodeFinder are connected by an associative network with weighted links to reflect the semantic relationships among components. Searching relevant components is supported by spreading activation. Frames in LaSSIE are structured into hierarchical, taxonomic categories by human experts. The multiple faceted classification scheme (Prieto-Diaz, 1991) is another format of structured representations in which reusable components are represented with multiple facets, each of which is described with a term. A conceptual distance graph has to be constructed to reflect the semantic relationships among these terms. AIRS is a system that combines multiple facets and the frame-based approach (Ostertag, Hendler et al., 1992). Structured representation-based systems are labor intensive in creating representations of components and knowledge bases.

Constraint-Based Reuse Repository Systems. Constraints of programs can also be used to index and retrieve reusable components. Rittri first proposed using signatures in reusable component retrieval (Rittri, 1989). His work was further extended by Zaremski and Wing, who presented a general framework for signature matching in functional programming languages (Zaremski and Wing, 1995). Research on signature matching has largely focused on functional programming languages that are often designed with a sound type theory. *CodeBroker* applies this technique to the strong-typed object-oriented programming language. Signature matching in *CodeBroker* is not used as the sole method of retrieving components; rather, it is used as a filter to exclude those components that are significantly different from the current task in terms of constraint compatibility.

The formal specification-based approach is another form of using constraints to index and retrieval components. Zaremski and Wing have adopted pre- and post-predicates to find components that exactly match or approximately match a reuse query (Zaremski and Wing, 1997). A. Mili and colleagues have tried to classify reusable components based on refinement order existing among their formal specifications (Mili, Mili et al., 1997). The formal specification-based approach could be integrated into *CodeBroker* to improve the precision of retrieval if the

programming environment supports formal methods. For the majority of programmers, however, the formal approach is too difficult to use.

Code-Based Reuse Repository Systems. Behavior sampling exploits the code aspect of programs to retrieve reusable components (Podgurski and Pierce, 1993). In behavior sampling-based systems, a programmer randomly chooses a small set of sample inputs and computes the corresponding outputs after having specified the signature of the module. Reusable components with compatible signatures are found and executed on the sample inputs. Components with outputs that match the outputs computed by the programmer are returned. Behavior sampling is difficult to apply to components with complex data structures, and it is unable to find close but not identical components.

8. SUMMARY

Locating components from a large reuse repository is the first step to the success of software reuse. However, passive reuse repository systems that rely on user-initiated browsing and searching mechanisms to locate components are only *hygienic factors* (Gellerman, 1963) for successful systematic reuse: They are prerequisites for effective motivation to reuse but by themselves are powerless to motivate software developers to reuse. To motivate software developers to reuse, we have tried to deal systematically with the cognitive and social challenges of software reuse by creating software development environments that are conducive to reuse. In this paper, we argued and demonstrated through the design, development, and evaluation of the *CodeBroker* system that reuse-conducive development environments based on the information delivery mechanism hold the potential of (1) making unanticipated components accessible to software developers, (2) reducing the overall cost of software reuse, and (3) motivating software developers to take a design approach that favors reuse by augmenting their knowledge of components. The challenge in implementing information delivery is to capture from the workspace as much information as possible to locate task-relevant and personalized information. In our research, we have tried to address the challenge by exploring (1) doc comments and signatures of the programs on which software developers are working, (2) discourse models that describe partially the overall goal of the development task, and (3) user models that represent the background knowledge of developers. We have demonstrated the feasibility of this approach with an implemented system. The evaluation of the system has shown its success in promoting software reuse in controlled experiments. We are currently conducting more experiments in natural settings to increase our understanding of the benefits and problems associated with our approach.

The major contribution of our research on reuse-conducive development environments is to explore and demonstrate the possibility of incorporating the information delivery mechanism into reuse repository systems. The information delivery mechanism is not meant to replace existing browsing and searching methods, but to complement them; and it has proven useful for cases in which software developers do not anticipate the existence of components or do not

know how to access them with browsing and searching. We believe other software reuse repository systems, especially those systems that are based on information retrieval methods, could benefit from our research results by including the support of delivery mechanisms in addition to their current support of searching.

CodeBroker represents a major step forward in our ongoing research framework of creating and evolving reuse repositories by enabling the active participation of application developers with the support of reuse-conducive development environments that address simultaneously the technical, cognitive, and social issues in software reuse. Software reuse cannot be truly successful until all three dimensions are properly supported.

ACKNOWLEDGMENTS

The authors would like to thank the members of the Center for LifeLong Learning & Design at the University of Colorado, who have made major contributions to the conceptual framework and systems described in this paper. The research was supported by (1) the National Science Foundation, Grants (a) REC-0106976 "Social Creativity and Meta-Design in Lifelong Learning Communities", and (b) CCR-0204277 "A Social-Technical Approach to the Evolutionary Construction of Reusable Software Component Repositories"; (2) the Ministry of Education, Culture, Sports, Science and Technology of Japan, Grant 15103 of the Open Competition for the Development of Innovative Technology program; (3) SRA Key Technology Laboratory, Inc., Tokyo, Japan; and (4) the Coleman Initiative, San Jose, CA.

9. REFERENCES

Alexander, C. (1964). *The Synthesis of Form*. Cambridge, MA: Harvard University Press.

Anquetil, N. and T. Lethbridge (1998). Extracting Concepts from File Names: A New File Clustering Criterion. *Proceedings of 20th International Conference on Software Engineering (ICSE'98)*, Kyoto, Japan, pp. 84-93.

Basili, V., L. Briand, et al. (1996). How Reuse Influences Productivity in Object-Oriented Systems. *Communications of the ACM*, **39**(10): 104-116.

Basili, V. R., R. W. Selby, et al. (1986). Experimentation in Software Engineering. *IEEE Transactions on Software Engineering*, **SE-12**(7): 733-743.

Belkin, N. J. (2000). Helping People Find What They Don't Know. *Communications of the ACM*, **43**(8): 58-61.

Brooks, F. P. J. (1995). *The Mythical Man-Month: Essays on Software Engineering, 20th Anniversary edition*. Reading, MA: Addison-Wesley.

Brown, J. S. and P. Duguid (2000). *The Social Life of Information*. Boston, MA: Harvard Business School Press.

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

DiFelice, P. and G. Fonzi (1998). How to Write Comments Suitable for Automatic Software Indexing. *Journal of Systems and Software*, **42**: 17-28.

Drummond, C., D. Ionescu, et al. (2000). A Learning Agent that Assists the Browsing of Software Libraries. *IEEE Transactions on Software Engineering*, **26**(12): 1179-1196.

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

Fafchamps, D. (1994). Organizational Factors and Reuse. *IEEE Software*, **11**(5): 31-41.

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

Fischer, G. (1987b). A Critic for LISP. *Proceedings of the 10th International Joint Conference on Artificial Intelligence*, Los Altos, CA, pp. 177-184.

Fischer, G. (1998). Seeding, Evolutionary Growth and Reseeding: Constructing, Capturing and Evolving Knowledge in Domain-Oriented Design Environments. *Automated Software Engineering*, **5**(4): 447-464.

Fischer, G. (2001). User Modeling in Human-Computer Interaction. *User Modeling and User-Adapted Interaction*, **11**(1&2): 65-86.
Fischer, G. (2002). Beyond "Couch Potatoes": From Consumers to Designers and Active Contributors. *First Monday (Peer-Reviewed Journal on the Internet)*, **7**(12): http://firstmonday.org/issues/issue7_12/fischer/.
Fischer, G., S. Henninger, et al. (1991). Cognitive Tools for Locating and Comprehending Software Objects for Reuse. *Proceedings of 13th International Conference on Software Engineering (ICSE'91)*, Austin, TX, pp. 318-328.
Fischer, G., A. C. Lemke, et al. (1985). Knowledge-Based Help Systems. In *Human Factors in Computing Systems (CHI'85)* San Francisco, CA, pp. 161-167.
Fischer, G., K. Nakakoji, et al. (1998). Embedding Critics in Design Environments. In *Readings in Intelligent User Interfaces*. M. T. Maybury and W. Wahlster, (Eds.) San Francisco, CA: Morgan Kaufman, pp. 537-561.
Fischer, G., E. Scharff, et al. (2003). Fostering Social Creativity by Increasing Social Capital. In *Social Capital*. M. Huysman and V. Wulf, (Eds.), (in press).
Fischer, G. and Y. Ye (2001). Personalizing Delivered Information in a Software Reuse Environment. *Proceedings of 8th International Conference on User Modeling*, Sonthofen, Germany, pp. 178-187.
Frakes, W. B. and C. J. Fox (1995). Sixteen Questions about Software Reuse. *Communications of the ACM*, **38**(6): 75-87.
Frakes, W. B. and C. J. Fox (1996). Quality Improvement Using a Software Reuse Failure Modes Model. *IEEE Transactions on Software Engineering*, **22**(4): 274-279.
Frakes, W. B. and T. P. Pole (1994). An Empirical Study of Representation Methods for Reusable Software Components. *IEEE Transactions on Software Engineering*, **20**(8): 617-630.
Furnas, G. W., T. K. Landauer, et al. (1987). The Vocabulary Problem in Human-System Communication. *Communications of the ACM*, **30**(11): 964-971.
Gamma, E., R. Johnson, et al. (1994). *Design Patterns—Elements of Reusable Object-Oriented Systems*. Reading, MA: Addison-Wesley.
Gellerman, S. W. (1963). *Motivation and Productivity*. New York: Amacom.
Griss, M. L. (2000). Implementing Product-Line Features with Component Reuse. *Proceedings of the 6th International Conference on Software Reuse (ICSR6)*, Vienna, Austria, pp. 137-152.
Grudin, J. (1994). Groupware and Social Dynamics: Eight Challenges for Developers. *Communications of the ACM*, **37**(1): 92-105.
Harman, D. (1995). Overview of the Third REtrieval Conference (TREC-3). In *The Third REtrieval Conference*. D. Harman, (Ed.) Gaithersburg, MD: National Institute of Standards and Technology Special Publication, pp. 1-19.
Henninger, S. (1997). An Evolutionary Approach to Constructing Effective Software Reuse Repositories. *ACM Transactions on Software Engineering and Methodology*, **6**(2): 111-140.
Isoda, S. (1995). Experiences of a Software Reuse Project. *Journal of Systems and Software*, **30**: 171-186.
Joos, R. (1994). Software Reuse at Motolora. *IEEE Software*, **11**(5): 42-47.
Karat, J., C.-M. Karat, et al. (2000). Affordances, Motivation, and the Design of User Interfaces. *Communications of the ACM*, **43**(8): 49-51.
Kintsch, W. (1998). *Comprehension: A Paradigm for Cognition*. Cambridge, UK: Cambridge University Press.
Landauer, T. K. and S. T. Dumais (1997). A Solution to Plato's Problem: The Latent Semantic Analysis Theory of Acquisition, Induction and Representation of Knowledge. *Psychological Review*, **104**(2): 211-240.
Lange, B. M. and T. G. Moher (1989). Some Strategies of Reuse in an Object-oriented Programming Environment. *Proceedings of Human Factors in Computing Systems*, Austin, TX, pp. 69-73.
Lieberman, H. (1997). Autonomous Interface Agents. *Proceedings of Human Factors in Computing Systems (CHI'97)*, Altanta, GA, pp. 67-74.
Lieberman, H., C. Fry, et al. (2001). Exploring the Web with Reconnaisssance Agents. *Communications of the ACM*, **44**(8): 69-75.
Maarek, Y. S., D. M. Berry, et al. (1991). An Information Retrieval Approach for Automatically Constructing Software Libraries. *IEEE Transactions on Software Engineering*, **17**(8): 800-813.
Maletic, J. I. and A. Marcus (2001). Supporting Program Comprehension Using Semantic and Structural Information. *Proceedings of 23rd International Conference on Software Engineering (ICSE'01)*, Toronto, Canada, pp. 103-112.
Michail, A. and D. Notkin (1999). Assessing Software Libraries by Browsing Similar Classes, Functions and Relationships. *Proceedings of 21st International Conference on Software Engineering (ICSE'99)*, Los Angeles, CA, pp. 463-472.
Mili, A., R. Mili, et al. (1997). Storing and Retrieving Software Components: A Refinement-Based System. *IEEE Transactions on Software Engineering*, **23**(7): 445-460.
Mili, A., R. Mili, et al. (1998). A Survey of Software Reuse Libraries. In *Systematic Software Reuse*. W. Frakes, (Ed.) Bussum, The Netherlands: Baltzer Science, pp. 317-347.
Mili, A., S. Yacoub, et al. (1999). Toward an Engineering Discipline of Software Reuse. *IEEE Software*, **16**(5): 22-31.

Nahapiet, J. and S. Ghoshal (1998). Social Capital, Intellectual Capital, and the Organizational Advantage. *Academy of Management Review*, **23**: 242-266.
Nardi, B. A. (1993). *A Small Matter of Programming*. Cambridge, MA: The MIT Press.
Nardi, B. A., J. R. Miller, et al. (1998). Collaborative, Programmable Intelligent Agents. *Communications of the ACM*, **41**(3): 96-104.
Ostertag, E., J. Hendler, et al. (1992). Computing Similarity in a Reuse Library System: An AI-Based Approach. *ACM Transactions on Software Engineering and Methodology*, **1**(3): 205-228.
Owen, D. (1986). Answers First, Then Questions. In *User Centered System Design, New Perspectives on Human-Computer Interaction*. D. A. Norman and S. W. Draper, (Eds.) Hillsdale, NJ: Erlbaum, pp. 361-375.
Podgurski, A. and L. Pierce (1993). Retrieving Reusable Software by Sampling Behavior. *ACM Transactions on Software Engineering and Methodology*, **2**(3): 286-303.
Poulin, J. S. (1999). Reuse: Been There, Done That. *Communications of the ACM*, **42**(5): 98-100.
Prieto-Diaz, R. (1991). Implementing Faceted Classification for Software Reuse. *Communications of the ACM*, **34**(5): 88-97.
Prieto-Diaz, R. (1996). Reuse as a New Paradigm for Software Development. In *Systematic Reuse: Issues in Initiating and Improving a Reuse Program*. M. Sarshar, (Ed.): Springer, pp. 1-13.
Raymond, E. S. and B. Young (2001). *The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*. Sebastopol, CA: O'Reilly.
Revelle, W. (1993). Individual Differences in Personality and Motivation: 'Non-Cognitive' Determinants of Cognitive Performance. In *Attention: Selection, Awareness and Control: A Tribute to Donald Broadbrent*. A. Baddely and L. Weiskrantz, (Eds.) Oxford: Oxford University Press, pp. 346-373.
Rhodes, B. J. and P. Maes (2000). Just-in-time Information Retrieval Agents. *IBM Systems Journal*, **39**(3&4): 685-704.
Rittel, H. (1984). Second-Generation Design Methods. In *Developments in Design Methodology*. N. Cross, (Ed.) New York: Wiley, pp. 317-327.
Rittri, M. (1989). Using Types as Search Keys in Function Libraries. *Journal of Functional Programming*, **1**(1): 71-89.
Robbins, J. E. and D. F. Redmiles (1998). Software Architecture Critics in the Argo Design Environment. *Knowledge-Based Systems*, **11**: 47-60.
Salomon, G., Ed. (1993). *Distributed Cognitions: Psychological and Educational Considerations*. Cambridge, United Kingdom: Cambridge University Press.
Salton, G. and M. J. McGill (1983). *Introduction to Modern Information Retrieval*. New York: McGraw-Hill.
Schmidt, D. C. (1999). Why Software Reuse Has Failed and How to Make It Work for You. *C++ Report*. **11**(1).
Sen, A. (1997). The Role of Opportunism in the Software Design Reuse Process. *IEEE Transactions on Software Engineering*, **23**(7): 418-436.
Simon, H. A. (1996). *The Sciences of the Artificial, Third edition*. Cambridge, MA: The MIT Press.
Soloway, E. and K. Ehrlich (1984). Empirical Studies of Programming Knowledge. *IEEE Transactions on Software Engineering*, **SE-10**(5): 595-609.
Terveen, L. G. (1995). An Overview of Human-Computer Collaboration. *Knowledge-Based Systems*, **8**(2-3): 67-81.
Tracz, W. (1990). The 3 Cons of Software Reuse. *Proceedings of 3rd Annual Workshop on Institutionalizing Software Reuse*, Syracuse, NY.
Visser, W. (1990). More or Less Following a Plan during Design: Opportunistic Deviations in Specification. *International Journal of Man-Machine Studies*, **33**(3): 247-278.
Williams, M. D. (1984). What Makes RABBIT Run? In *International Journal of Man-Machine Studies*. **21**, pp. 333-352.
Ye, Y. (2001). *Supporting Component-Based Software Development with Active Component Repository Systems.* Ph.D. Dissertation, Department of Computer Science, University of Colorado, Boulder, CO, available at http://www.cs.colorado.edu/~yunwen/thesis/.
Ye, Y. and G. Fischer (2002). Information Delivery in Support of Learning Reusable Software Components on Demand. *Proceedings of 2002 International Conference on Intelligent User Interfaces (IUI'02)*, San Francisco, CA, pp. 159-166.
Ye, Y., G. Fischer, et al. (2000). Integrating Active Information Delivery and Reuse Repository Systems. *Proceedings of ACM SIGSOFT 8th International Symposium on Foundations of Software Engineering (FSE8)*, San Diego, CA, pp. 60-68.
Ye, Y. and K. Kishida (2003). Toward an Understanding of the Motivation of Open Source Software Developers. *Proceedings of 2003 International Conference on Software Engineering (ICSE'03)*, Portland, OR, pp. 419-429.
Zaremski, A. M. and J. M. Wing (1995). Signature Matching: A Tool for Using Software Libraries. *ACM Transactions on Software Engineering and Methodology*, **4**(2): 146-170.
Zaremski, A. M. and J. M. Wing (1997). Specification Matching of Software Components. *ACM Transactions on Software Engineering and Methodology*, **6**(4): 333-369.