Proceedings of 2002 International Conference on Software Engineering (ICSE'02), Orlando, FL, pp 513-523, May 19-25, 2002

Supporting Reuse by Delivering Task-Relevant and Personalized Information

Yunwen Ye[1,2]
[1]SRA Key Technology Laboratory, Inc.
3-12 Yotsuya, Shinjuku, Tokyo 160-004, Japan
+1-303-492-8136
yunwen@cs.colorado.edu

Gerhard Fischer[2]
[2]Department of Computer Science
University of Colorado
Boulder, CO80303-0430, USA
+1-303-492-1592
gerhard@cs.colorado.edu

ABSTRACT

Technical, cognitive, and social factors inhibit the widespread success of systematic software reuse. Our research is primarily concerned with the cognitive and social challenges faced by software developers: how to motivate them to reuse and how to reduce the difficulty of locating components from a large reuse repository. Our research has explored a new interaction style between software developers and reuse repository systems enabled by information delivery mechanisms. Instead of passively waiting for software developers to explore the reuse repository with explicit queries, information delivery autonomously locates and presents components by using the developers' partially written programs as implicit queries.

We have designed, implemented, and evaluated a system called *CodeBroker*, which illustrates different techniques to address the essential challenges in information delivery: *to make the delivered information relevant to the task-at-hand and personalized to the background knowledge of an individual developer*. Empirical evaluations of *CodeBroker* show that information delivery is effective in promoting reuse.

1. INTRODUCTION

Although it is widely believed that software reuse improves both the quality and productivity of software development [2], 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 initial investments, both financially and intellectually;
- Enabling software developers to build new software systems with components from the reuse repository.

The above 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 break this deadlock is to focus on the creation of a good reuse repository first and then to institute a reuse program top-down by enforcing reuse through education and other organizational changes [7]. 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 [19]. Such an approach requires *reuse-conducive* development environments that can reduce the challenges faced by developers who reuse software components and contribute to reuse repositories [45].

Creating reuse-conducive development environments poses not only technical challenges, such as effective tools to help software developers locate and understand components, but also cognitive and social challenges, such as what motivates software developers to initiate the reuse process and contribute to the reuse repository, and what the difficulties are in such activities [33].

Our research is primarily concerned with the cognitive and social challenges of reuse [8, 46]. Software developers are unable, or unwilling, to reuse if they do not know the existence of reusable components or do not know how to locate, understand, and use them.

Many reuse research efforts [27, 29, 32] have tried to design various mechanisms and tools to assist software developers in locating and understanding components. Browsing and searching are the principal mechanisms used to locate components. In both mechanisms, software developers must initiate and operate the locating process; therefore, the success of reuse is greatly affected by how well they know about the reuse repository [4]. Browsing and searching are of little use for less experienced software developers because they do not even anticipate the existence of components or do not know how to use the reuse repository properly. Large reuse repositories that contain many components are essential for the widespread success of systematic reuse. As reuse repositories grow larger, it becomes increasingly difficult for software developers to know or anticipate the existence of components. We propose here a new mechanism for locating components: *information delivery* that autonomously locates and presents software developers with task-relevant and personalized components. We have designed and implemented a system called *CodeBroker*, which illustrates different techniques to make the delivered components relevant to the task-at-hand and personalized to the background knowledge of an individual developer. Empirical evaluations of *CodeBroker* show that information delivery is effective in promoting reuse.

2. OUR CONCEPTUAL FRAMEWORK

Reuse repository systems are a subset of *high-functionality applications* (HFAs) [9] 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 their current task (Figure 1).

Our empirical studies [9] 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 different levels of knowledge of the information space. L1 represents the elements that are well known and can be easily used by users, even without consulting help and documentation systems. L2 contains the elements that users know vaguely. L3 contains elements that users anticipate to exist in the HFA. A portion of L3 lies outside the actual information space, so it contains the elements that the user believes exist but 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 information retrieval, human-computer interaction, 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, not only in the phase of implementing software systems, but also in the phases of requirement analysis and system design because they can map concepts in problem domains directly into reusable components that have higher abstraction levels [20, 22]. Such a reuse approach is often referred to as "opportunistic reuse" [38] 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. To do so, they have to incorporate a reuse process into their current development process. A reuse process consists of three steps: locating the components reusable in the task-at-hand, comprehending the functionality and usage of the components, and modifying the components if there is not any component that completely fits the task [11].

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" [15] is often regarded as an attitude problem, and is commonly labeled as the "Not-Invented-Here" syndrome. Many empirical studies [14, 20, 25] have shown, however, that software developers would put a lot of effort into locating and reusing components if they knew of the components that can 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 [8, 46]. 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;
- may perceive that reuse costs more than developing from scratch;
- may not be able to use the repository system by formulating a proper query or browsing the repository to locate components in L3; and
- may not be able to understand the found components.

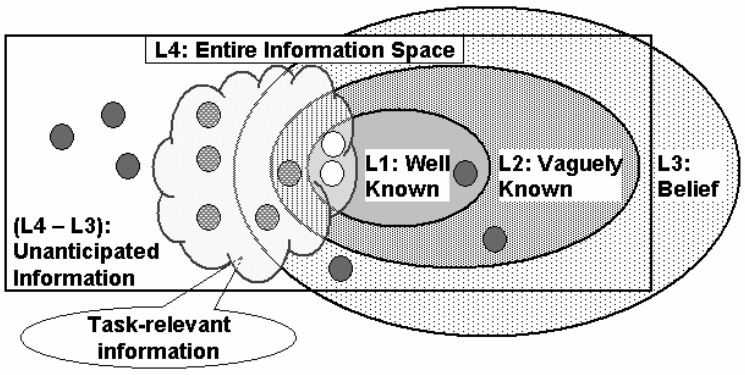

Figure 1: Different levels of knowledge about a high-functionality application (HFA)

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 3.3).

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 [37]. Various schemas of indexing and retrieving software components have been proposed in previous reuse research [16]. Although such schemas are very important, it is equally, if not more, important to investigate 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 [21]. 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 may want to draw a circle, but she must know that the method is called `drawOval` in the Java class library to search it; or she must know that this method is in the `java.awt` package and in the `Graphics` class if she prefers browsing. This *conceptual gap* between situation and system models is a significant cognitive barrier to locating components [11].

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 information locating process, information delivery infers the needs for information by monitoring the low-level activities of users, and autonomously locates and delivers information based on the inferred needs [31]. Information delivery is needed to take advantage of the large number of potentially useful components contained in (L4 – L3) (see Figure 1). The big challenge in making information delivery systems useful is to exploit the working context and the distinct information needs of each user to present only those components that are related to the task-at-hand and are not yet known to the individual user [13], rather than bombarding users with decontextualized and irrelevant information. An example of a decontextualized information delivery system that almost all users choose not to use is Microsoft Office's *Tip of the Day* [9].

Retrieval-by-Reformulation. Because of the aforementioned conceptual gap and the 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. *Retrieval-by-reformulation* [44] is the process that allows users to incrementally improve their query after they have familiarized themselves with the information space by evaluating previous retrieval results. Retrieval-by-reformulation is especially important in information delivery systems that infer information needs. Combining information delivery and retrieval-by-reformulation makes the information location process a collaborative one in which computers and users complement each other's strengths.

2.3 Human-Computer Interaction and Reuse

A fundamental objective of human-computer interaction in an information-rich world is to provide computer users with experiences that fit their specific background knowledge and tasks. The challenge in implementing reuse repository systems is not only to make components available to software developers, but to locate the "right" component (task-relevant) at the "right" time (when it is needed) to the "right" software developer (personalized).

Multiple Communication Channels. Traditionally, information needs are communicated to systems through a narrow *explicit communication channel* established by users when they start browsing or searching. Because software developers work with computers, reuse repository systems can explore the power of *implicit communication channels* [9] that can be established when reuse repository systems are integrated with development environments [12]. Such an integration makes the workspace of software developers accessible to reuse repository systems that can infer the needs for software components from partially written programs. Based on the inferred needs, reuse repository systems can deliver task-relevant components without explicit queries from users.

Personalized Information Needs. Because different software developers have differing knowledge about the reuse repository, reuse repository systems should not return the same set of components to all software developers. To personalize the located components to the specific background of each software developer requires *user models* [9] to represent the existing knowledge that software developers have of the reuse repository. User models can be adaptive and adaptable. User models are *adaptive* if the system implicitly creates and updates them by observing the interactions between the system and users; and they are *adaptable* if users explicitly update them by adding or removing information.

2.4 Knowledge-Based Systems and Reuse

The influence of knowledge-based systems on reuse is twofold. First, reuse repository systems are software developers' assistants that 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. Cognitive theory has 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 [39]. That explains why software developers with differing knowledge often choose very different approaches to develop the same task [42]. 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 the 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 decompose the task further [38].

Information delivery can make unknown components reused in a way similar to known components. Because timely delivered components based on the 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 they are known or not, may possibly actively contribute to the software development process.

Finding Task-Relevant Components with Similarity Analysis. The two basic approaches to infer the high-level goals of users from their low-level activities and then find 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. The logic assumption of similarity analysis is: "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 programs [1, 29, 40]; *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 [47]. Therefore, the relevance of a component to the task-at-hand can be determined by the *conceptual similarity* between the comments 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 (LSA). LSA [24] 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 [21], and therefore has the potential to reduce the conceptual gap between situation model and system model in locating components.

3. DELIVERING TASK-RELEVANT AND PERSONALIZED COMPONENTS

Based on our conceptual framework, we have designed, implemented, and evaluated a system called *CodeBroker* that delivers task-relevant and personalized components. It supports Java developers in reusing components without leaving their development environment, *Emacs*, which is augmented with the *RCI-display* (Reusable Component Information display, the lower buffer in Figure 2), where task-relevant and personalized components are autonomously shown.

3.1 Overview of the CodeBroker System

CodeBroker (Figure 3) consists of an *interface agent* and a back-end search engine. Running continuously as a background process in *Emacs*, the interface agent infers and extracts reuse queries by monitoring development activities. Queries are passed to the search engine, which retrieves matching components. Retrieved components are delivered by the interface agent in the *RCI-display*, after it has removed the components that are contained in discourse models and user models. *Discourse models*, created by software developers during previous interactions with the system, include components that these developers have indicated are of no interest in the current development session. *User models*, created and updated by both the system and software developers, contain components known to individual software developers.

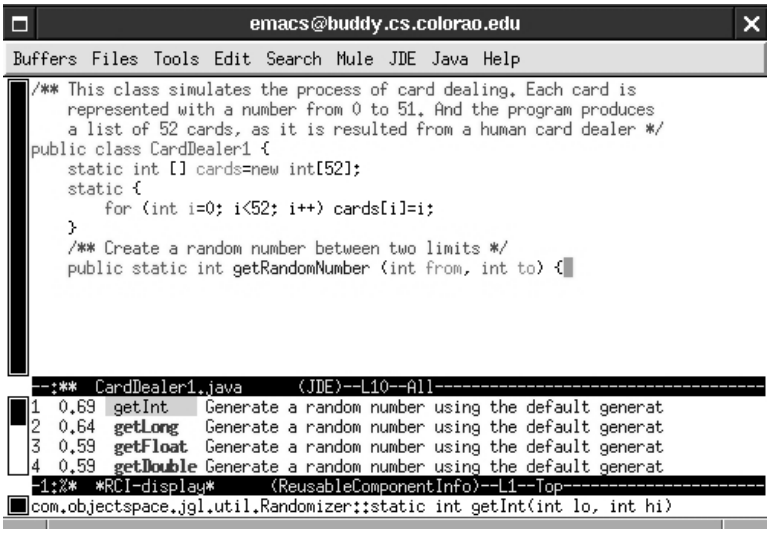

Figure 2: An example of the use of *CodeBroker*

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, and describes the task in the doc comment and signature before the cursor, based on which 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.

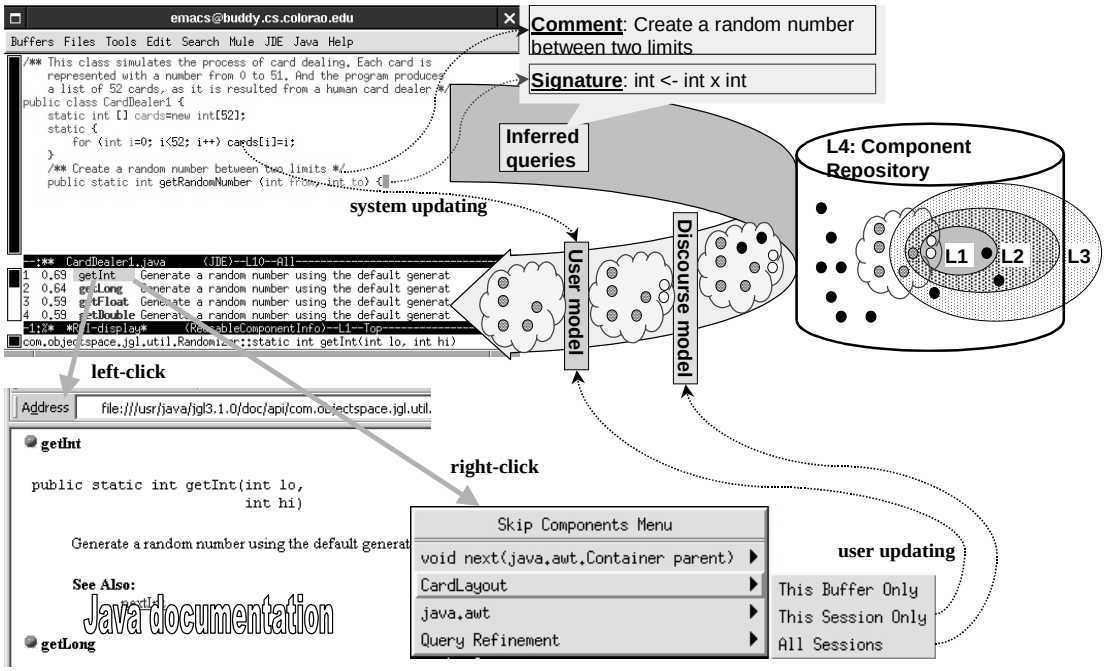

Figure 3: The system architecture of *CodeBroker*

Components that match the queries, which are extracted from doc comments and signatures, are delivered after being filtered with discourse models and user models. Discourse models (see Section 3.2.3) remove irrelevant components (black dots), and user models (see Section 3.3) remove known components (unshaded dots). Discourse models and user models can both be updated by users through the `Skip Components Menu`. User models are also automatically updated when the system detects the reuse of a component in the workspace. Users who want to know more about a component can go to the Java documentation by clicking on the delivered component.

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.

CodeBroker delivers components whenever a doc comment or a signature definition is entered. For example, in Figures 2 and 3, the developer who wants to create a random number between two integers writes a doc comment. As soon as the rightmost '/' (signaling the end of a doc comment) is entered, the contents of the doc comment are extracted as a query, and components from the repository that match it are shown immediately in the *RCI-display*. Because there are many random number generators that operate on different data types, the software developer may want to find the one that takes integers as input. 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), *CodeBroker* extracts the signature, which is then combined with the preceding doc comment as a query to retrieve matching components. The first component in the *RCI-display* in Figure 2 does exactly what the developer wants and can be reused immediately.

CodeBroker presents information with three different layers of abstraction. The first layer is the *RCI-display* in which 20 (the number can be customized) components are shown according to their task relevance, and each component is accompanied by its rank of relevance, relevance value, name, and synopsis. To reduce the intrusiveness [12], users are not required to interact with the system if they are not interested in the delivered components. If they 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 2); 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 3).

If the software developer feels too many irrelevant components are delivered in the *RCI-display*, activating the `Skip Components Menu` associated with each component will filter them out (Figure 3). 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 3.2.3); and the third command, `All Sessions`, both removes the chosen item from the buffer and adds it to the user model (see Section 3.3).

3.2 Delivering Task-Relevant Components

CodeBroker explores multiple communication channels to deliver task-relevant components. Through the *implicit communication channel* established by its interface agent, it autonomously extracts reuse queries from partially written programs. It uses the discourse models created by software developers through an *explicit communication channel* to capture the larger context of the development task.

3.2.1 Extracting Queries Autonomously

Reuse queries are extracted from doc comments and signatures. 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 [41], 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 [40], which are important beacons to understanding programs [1, 28, 29]. 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 is going to be incorporated.

Based on the assumption of *similarity analysis*, a component is highly likely to be reused if it shows either conceptual similarity, or constraint compatibility, or both, to the task-at-hand. The conceptual similarity that exists between the textual document of a component and the doc comment extracted from *Emacs* is determined by LSA. The constraint compatibility, or signature compatibility, that exists between the signature of a component and the extracted signature, is determined by the process of signature matching [46].

3.2.2 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 the task-relevant information only [37]. *CodeBroker* unavoidably delivers some irrelevant components and misses some relevant components. The *retrieval-by-reformulation* interface (Figure 4) in *CodeBroker* enables software developers to incrementally reformulate reuse queries after they have studied the delivered components, until they locate what they want.

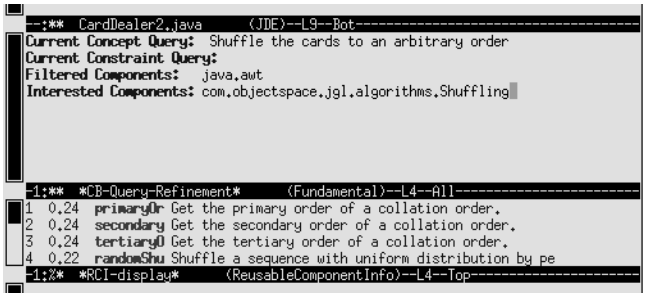

Figure 4: The retrieval-by-reformulation interface

The retrieval-by-reformulation interface represents an *explicit communication channel* that must be activated by users, and through which they can refine queries and also 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, they can exclude components from certain packages and classes by adding their names to the `Filtered Components` field, or limit the search to packages and classes of interest by adding their names to the `Interested Components` field (Figure 4).

Although the interface can also be used as the 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. The delivered components can familiarize them with the repository and enable them over time to formulate reuse queries that are closer to the *system model*.

3.2.3 Creating 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.

```
;; Discourse model for the session started at Thu Nov 2 08:53:12
((("java.util.zip") ;; Package added at Thu Nov 2 08:55:53.
  ("java.awt" ("CardLayout"))) ;; Class added at Thu Nov 2 09:20:3
```

Figure 5: An example discourse model

Each development session starts with an empty discourse model that is incrementally updated by the software developer as he 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 users to identify misfits than fits. Figure 5 shows an example of a discourse model extracted from one of our experiments, in which the subject was asked to create a program that simulates the process of dealing cards randomly. *CodeBroker* responded to his initial doc comments with some components from the package `java.util.zip` and the class `java.awt.CardLayout` because their documents contained some words that also appeared in the doc comments. The subject knew he did not need any components from the package and the class for his task, so he right-clicked on the components to launch the `Skip Components Menu` (Figure 3) to add the package and the class to the discourse model. For the

remaining interactions with the system in the same development session, all components from the package and the class were not delivered to the subject.

Both a discourse model and the `Filtered Components` field in the retrieval-by-reformulation interface (Figure 4) 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 only used 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.

3.3 Delivering Personalized Components

Delivering a component that is well known to a software developer is not desirable. Because each software developer has different knowledge about the reuse repository, the system needs to personalize its delivery to each developer's unique 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 6: 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 6) to represent software developers' knowledge about the reuse repository. User models contain both well-known (L1 in Figure 1) and vaguely known (L2) components. Only well-known components are removed from deliveries 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. Software developers can explicitly adapt their user models. When a known component is delivered and they do not want the same component to be delivered again, they can use the `Skip Components Menu` (Figure 3) to add the component, its class, or its package to user models. Such user-added components do not have a use-time field in user models, and 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 (Figure 3). It uses heuristic rules 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 the 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 components added to user models by the system have a use time, which is the time the component is detected to be invoked in the editor. Components that have more than three use times (the number is customizable) are considered as well known, namely, included in L1 (Figure 1), and components that have fewer than four use times are considered as vaguely known (L2).

4. EVALUATION

To understand how information delivery can support component reuse, we have conducted empirical evaluations of *CodeBroker* with software developers.

The reuse repository used in evaluation experiments included 673 classes and 7,338 methods from the Java 1.1.8 core library and JGL 1.3 library (created by Objectspace, Inc.). The semantic space created by LSA was trained with documents from four sources: Linux on-line manuals, programming textbooks, the Java language specification and virtual machine specification, and Java class libraries. In total, there were 78,475 documents and 10,988 different terms.

4.1 Recall and Precision

Information retrieval systems are conventionally evaluated by recall and precision [37]. 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 7 shows the recall-precision curve for the results of executing 19 queries in *CodeBroker*. A half of the queries were created by us, and the other half were collected from empirical experiments and frequently asked questions in Java-related newsgroups. The data shown in Figure 7 is lower than those reported in the evaluations of other reuse repository systems [16]. However, retrieval systems can be compared only when all the

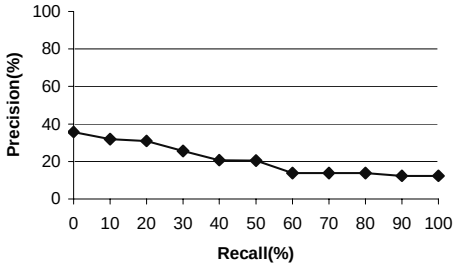

Figure 7: The recall-precision curve

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.

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. 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 [3].

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. 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 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 1 shows the overall results of the experiments. Subjects reused delivered components during 10 of the 12 experiments. 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 1). In other words, those 9 components could not have been reused without the support of *CodeBroker*, and the subjects would have created their own solutions instead. 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."

Reducing Locating Time. Although the subjects anticipated the existence of the other 11 components (see 6th and 7th columns 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.

> "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 index of methods, it is hard to find here [the Java documentation system]."

Table 1: Overall Results of Experiments

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
				Unanticipated (L4-L3)	Anticipated but unknown (L3)	Vaguely known (L2)	
S1	1	10	4	2	2	0	0
	2	3	1	1	0	0	1
S2	3	7	1	1	0	0	0
	4	4	1	1	0	0	0
	5	5	3	0	2	1	1
S3	6	5	2	1	1	0	1
	7	4	3	1	2	0	1
	8	3	0	0	0	0	0
S4	9	4	3	0	3	0	0
	10	3	1	1	0	0	2
S5	11	4	1	1	0	0	2
	12	5	0	0	0	0	0
Sum		**57**	**20**	**9**	**10**	**1**	**8**

Snowball Effects of Deliveries. The last column in Table 1 shows the number of components that were not delivered but were triggered to be reused by deliveries. In some cases, when the subjects wanted to reuse a delivered component that requires other supplementary components, they had to find those components through browsing. Subjects had not known those triggered components before, and it was the deliveries that 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, 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 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 4). Subject S5, who did not even know the existence of the `java.text` package, described as his implementation plan that he was going "to parse the number, take out the commas and insert the commas". As 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 S3. Subject S2, who did not know the `java.text.NumberFormat` class either, described a plan like 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.

In the experiments, we observed several other occasions 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.

Roles of Discourse Models. Discourse models improved the task-relevance of delivered components when they were created. In five experiments, subjects created discourse models, which removed about 10% of retrieved components from the deliveries. All of the removed components were irrelevant to the task-at-hand.

Roles of User Models. The experiments, however, did not yield strong and conclusive data regarding the roles of user models. Only 2% of the retrieved components were filtered by user models. 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; or (2) to observe the effectiveness of delivering unknown components, subjects were assigned the tasks that involved the part of the repository they did not know very well, and, consequently, most delivered components were unknown. Nevertheless, user models helped and are 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.

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' capability 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 Table 2, on a scale from 1 (totally useless) to 10 (extremely useful).

Table 2: Subjective ratings on the usefulness of CodeBroker

Subject	S1	S2	S3	S4	S5
Rating	7	4	8.5	7	8

5. DISCUSSIONS AND FUTURE WORK

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 [31]. 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 7). 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 [17], 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 implementation level. This is important because this 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 other phases of the software engineering process.

We should be 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 very 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 a less respected source. To answer this question, we need to investigate the social aspects of software reuse, such as what makes software developers trust a component and how to involve them in the evolution of the reuse repository to complement, or even replace, a dedicated team of component developers [10, 33].

6. RELATED WORK

Most of the previous research on reuse repositories has focused on the indexing and retrieval mechanisms. Different mechanisms, such as free-text retrieval [27], multi-facets classification [32], semantic networks [5], spreading activation [19], behavior sampling [18], signature matching [47], and specification matching [30], have been proposed. *CodeBroker* combines both free-text retrieval based on LSA and signature matching in its retrieval mechanism.

Parts of *CodeBroker* are similar to the systems that use identifier names, comments, or both to cluster components that have similar functionality. Such clusters can help software developers choose reusable components [29] or comprehend existing software systems [1, 28].

The cliché-based programming environment *KBEmacs* [35] is also implemented as an extension to *Emacs*. It has a knowledge base of clichés that programmers can reuse. *KBEmacs* helps

programmers who already knew the cliché because programmers have to refer to it by name, whereas *CodeBroker* tries to give programmers access to unknown components.

Wren [23] is a component-based development environment that supports software developers in locating, evaluating, and incorporating components from several component distribution sites. It also stresses the importance of making use of self-revealing information contained in components. However, no automated location support is provided in *Wren*.

Reuse repository systems that support information delivery are autonomous interface agents [26] that proactively retrieve and deliver information by predicting the information needs of users. *Remembrance Agent* [34] continually presents, from the user's personal archive, a list of documents that are relevant to the current document being written. *Letizia* [26] assists users in browsing the WWW by suggesting and displaying relevant web pages based on user interests. By observing the programmer's Java programming, *Expert Finder* [43] can refer the programmer to expert helpers who have displayed significant experience in the area in which the programmer is troubled.

Information delivery has been explored in several other research prototypes of software development environments. Drummond et. al [6] add 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. The *Argo* design environment [36] is equipped with computer critics [12] that deliver general software design knowledge for software developers to reflect upon their current design.

7. CONCLUSIONS

Locating components from a large reuse repository is the first step to the success of software reuse. Information delivery holds the potential of (1) making unanticipated components easily 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 doc comments and signatures of the programs on which software developers are working, discourse models that describe partially the overall goal of the development task, and user models that represent the background knowledge of developers. We have demonstrated the feasibility of this approach with an implemented system. The empirical evaluations of the system have shown its success in promoting software reuse in controlled experiments. We are currently conducting more experiments in natural settings to further our understanding of the benefits and problems associated with our approach.

The unique contribution of our research is that it explores a new style of human-computer collaboration in software reuse, transcending the traditional interface of reuse repository systems that rely on the explicit communication channel established by software developers when they initiate the reuse process. The information delivery mechanism is not meant to replace the 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.

8. ACKNOWLEDGMENTS

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

9. REFERENCES

[1] Anquetil, N., and Lethbridge, T. Extracting Concepts from File Names: A New File Clustering Criterion, in Proceedings of 20th ICSE (Kyoto, Japan, 1998), 84-93.

[2] Basili, V., Briand, L., and Melo, W. How Reuse Influences Productivity in Object-Oriented Systems. Commun. ACM, 1996. 39(10): 104-116.

[3] Basili, V.R., Selby, R.W., and Hutchen, D.H. Experimentation in Software Engineering. IEEE Trans. on Software Engineering, 1986. SE-12(7): 733-743.

[4] Creech, M.L., Freeze, D.F., and Griss, M.L. Using Hypertext in Selecting Reusable Software Components, in Proceedings of Hypertext'91 (San Antonio, TX, 1991), 25-38.

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

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

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

[8] Fischer, G. Cognitive View of Reuse and Redesign. IEEE Software, 1987. 4(4): 60-72.

[9] Fischer, G. User Modeling in Human-Computer Interaction. User Modeling and User-Adapted Interaction, 2001. 11(1&2): 65-86.

[10] Fischer, G., et al. Seeding, Evolutionary Growth and Reseeding: The Incremental Development of Collaborative Design Environments, in Coordination Theory and Collaboration Technology, Olson, G., Malone, T., and Smith, J. (eds.), Lawrence Erlbaum, Mahwah, NJ, 2001, 447-472.

[11] Fischer, G., Henninger, S., and Redmiles, D. Cognitive Tools for Locating and Comprehending Software Objects for Reuse, in Proceedings of 13th ICSE (Austin, TX, 1991), 318-328.

[12] Fischer, G., et al. Embedding Critics in Design Environments. The Knowledge Engineering Review Journal, 1993. 8(4): 285-307.

[13] Fischer, G., and Ye, Y. Personalizing Delivered Information in a Software Reuse Environment, in Proceedings of 8th International Conference on User Modeling (Sonthofen, Germany, 2001), 178-187.

[14] Frakes, W.B., and Fox, C.J. Sixteen Questions about Software Reuse. Commun. ACM, 1995. 38(6): 75-87.

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

[16] Frakes, W.B., and Pole, T.P. An Empirical Study of Representation Methods for Reusable Software Components. IEEE Trans. on Software Engineering, 1994. 20(8): 617-630.

[17] Gamma, E., et al. Design Patterns--Elements of Reusable Object-Oriented Systems. Addison-Wesley, Reading, MA, 1994.

[18] Hall, R.J. Generalized Behavior-Based Retrieval, in Proceedings of 15th ICSE (Baltimore, MD, 1993), 371-380.

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

[20] Isoda, S. Experiences of a Software Reuse Project. Journal of Systems and Software, 1995. 30: 171-186.

[21] Kintsch, W. Comprehension: A Paradigm for Cognition. Cambridge University Press, Cambridge, UK, 1998.

[22] Krueger, C.W. Software Reuse. ACM Computing Surveys, 1992. 24(2): 131-183.

[23] Lüer, C., and Rosenblum, D.S. Wren--An Environment for Component-Based Development, in Proceedings of the Joint ESEC-8 and FSE-9 (Vienna, Austria, 2001), 207-217.

[24] Landauer, T.K., and Dumais, S.T. A Solution to Plato's Problem: The Latent Semantic Analysis Theory of Acquisition, Induction and Representation of Knowledge. Psychological Review, 1997. 104(2): 211-240.

[25] Lange, B.M., and Moher, T.G. Some Strategies of Reuse in an Object-oriented Programming Environment, in Proceedings of Human Factors in Computing Systems (Austin, TX, 1989), 69-73.

[26] Lieberman, H. Autonomous Interface Agents, in Proceedings of Human Factors in Computing Systems (Altanta, GA, 1997), 67-74.

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

[28] Maletic, J.I., and Marcus, A. Supporting Program Comprehension Using Semantic and Structural Information, in Proceedings of 23rd ICSE (Toronto, Canada, 2001), 103-112.

[29] Michail, A., and Notkin, D. Assessing Software Libraries by Browsing Similar Classes, Functions and Relationships, in Proceedings of 21st ICSE (Los Angeles, CA, 1999), 463-472.

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

[31] Nardi, B.A., Miller, J.R., and Wright, D.J. Collaborative, Programmable Intelligent Agents. Commun. ACM, 1998. 41(3): 96-104.

[32] Prieto-Diaz, R. Implementing Faceted Classification for Software Reuse. Commun. ACM, 1991. 34(5): 88-97.

[33] Raymond, E.S., and Young, B. The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary. O'Reilly, Sebastopol, CA, 2001.

[34] Rhodes, B.J., and Maes, P. Just-in-time Information Retrieval Agents. IBM Systems Journal, 2000. 39: 685-704.

[35] Rich, C.H., and Waters, R.C. The Programmer's Apprentice. Addison-Wesley, Reading, MA, 1990.

[36] Robbins, J.E., and Redmiles, D.F. Software Architecture Critics in the Argo Design Environment. Knowledge-Based Systems, 1998. 11: 47-60.

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

[38] Sen, A. The Role of Opportunism in the Software Design Reuse Process. IEEE Trans. on Software Engineering, 1997. 23(7): 418-436.

[39] Simon, H.A. The Sciences of the Artificial, Third edition. The MIT Press, Cambridge, MA, 1996.

[40] Soloway, E., and Ehrlich, K. Empirical Studies of Programming Knowledge. IEEE Trans. on Software Engineering, 1984. SE-10(5): 595-609.

[41] Tracz, W. The 3 Cons of Software Reuse, in Proceedings of 3rd Annual Workshop on Institutionalizing Software Reuse (Syracuse, NY, 1990).

[42] Visser, W. More or Less Following a Plan during Design: Opportunistic Deviations in Specification. International Journal of Man-Machine Studies, 1990. 33(3): 247-278.

[43] Vivacqua, A. Agents for Expertise Location, in Proceedings of 1999 AAAI Spring Symposium on Intelligent Agents in Cyberspace (Stanford, CA, 1999), 9-13.

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

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

[46] Ye, Y., Fischer, G., and Reeves, B. Integrating Active Information Delivery and Reuse Repository Systems, in Proceedings of FSE-8 (San Diego, CA, 2000), 60-68.

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