

Promoting Reuse with Active Reuse Repository Systems

Yunwen Ye^{1,2} and Gerhard Fischer¹

¹ Department of Computer Science, CB 430, University of Colorado at Boulder,
Boulder, CO. 80309-0430, USA

{yunwen, gerhard}@cs.colorado.edu

² Software Engineering Laboratory, Software Research Associates, Inc.
3-12 Yotsuya, Shinjuku, Tokyo 160-0004, Japan

Abstract. Software component-based reuse is difficult for software developers to adopt because first they must know what components exist in a reuse repository and then they must know how to retrieve them easily. This paper describes the concept and implementation of *active reuse repository systems* that address the above two issues. Active reuse repository systems employ active information delivery mechanisms to deliver potentially reusable components that are relevant to the current development task. They can help software developers reuse components they did not even know existed. They can also greatly reduce the cost of component location because software developers need neither to specify reuse queries explicitly, nor to switch working contexts back and forth between development environments and reuse repository systems.

1 Introduction

Component-based software reuse is an approach to build new software systems from existing reusable software components. Software reuse is promising because complex systems evolve faster if they are built upon stable subsystems [34]. Empirical studies have also concluded that software reuse can improve both the quality and productivity of software development [1, 23]. However, successful deployment of software reuse has to address managerial issues, enabling technical issues and cognitive issues faced by software developers. This paper is focused on the technical and cognitive issues involved in component-based software reuse.

A reuse process generally consists of location, comprehension, and modification of needed components [13]. As a precondition to the success of reuse, a reuse repository system is indispensable. A reuse repository system has three connotations: a collection of reusable components, an indexing and retrieval mechanism, and an operating interface. Reuse repository systems suffer from an inherent dilemma: the more components they include, the more potentially useful they are, but also the more difficult they become for reusers to locate the needed components. Nevertheless, for reuse to pay off, a reuse repository with a large number of components is necessary. The success of reuse thus relies crucially on

the retrieval mechanism and the interface of reuse repository systems to facilitate the easy location of components.

Existence of reusable components does not guarantee their being reused. Reusable components help software developers think at higher levels of abstraction. Like the introduction of a new word into English that increases our power in thinking and communication, reusable components expand the expressiveness of software developers and contribute to the reduction of the complexity of software development. However, software developers must learn the syntax and semantics of components if they are able to develop with them. Components learning constitutes no small part of the cognitive barrier to reuse [4].

Another truism in reuse is that for software developers to reuse, they must be able to locate reusable components easier than developing from scratch [20]. It is important to put reuse into the whole context of software development. For software developers, reuse is not their goal; reuse is only means for them to accomplish their tasks. Cognitive scientists have revealed that human beings are utility-maximizers [30], therefore only when software developers perceive that the reuse approach has more value than its cost, will reuse be readily embraced.

In order to diminish the above-mentioned two barriers to reuse faced by software developers, a developer-centered approach to the design of reuse repository systems is proposed in this paper. This approach stresses the importance of integrating reuse repository systems into the development environment and views reuse as an integral part of the development process. Drawing on empirical studies and cognitive theory, we first analyze the difficulties of reuse from the perspective of software developers in Sect. 2. We argue in Sect. 3 that active reuse repository systems—systems equipped with active information delivery mechanisms—are a solution because they increase developers’ awareness of reusable components, and reduce the cost of component location. A prototype of an active reuse repository system, *CodeBroker*, is described in Sect. 4.

2 Developer-Centered View of Reuse

2.1 Three Modes of Reuse

From the perspective of software developers, there are three reuse modes based upon their knowledge about a reuse repository: *reuse-by-memory*, *reuse-by-recall* and *reuse-by-anticipation*.

In the reuse-by-memory mode, while developing a new system, software developers may notice similarities between the new system and reusable components they have learned in the past and know very well. Therefore, they can reuse them easily during the development, even without the support of a reuse repository system because their memory assumes the role of the repository system.

In the reuse-by-recall mode, while developing a new system, software developers vaguely recall that the repository contains some reusable components with similar functionality, but they do not remember exactly which components they are. They need to search the repository to find what they need. In this mode,

developers are often determined to find the needed components. An effective retrieval mechanism is the main concern for reuse repository systems supporting this mode.

In the reuse-by-anticipation mode, software developers anticipate the existence of certain reusable components. Although they don't know of relevant components for certain, their knowledge of the domain, the development environment, and the repository is enough to motivate them to search in hopes of finding what they want from the reuse repository system. In this mode, if developers cannot find what they want quickly enough, they will soon give up reuse [26].

Software developers have little resistance to the first two modes of reuse. As is reported by Isoda [19], software developers reuse those components repeatedly once they have reused them once. This also explains why individual ad hoc reuse has been taking place while organization-wide systematic reuse has not received the same success: software developers have individual reuse repositories in their memories so they can reuse-by-memory or reuse-by-recall [26]. For those components that have not yet been internalized into their memories, software developers have to resort to the mode of reuse-by-anticipation. The activation of the reuse-by-anticipation mode relies on two enabling factors:

- Software developers anticipate the existence of reusable components.
- They perceive that the cost of the reuse process is cheaper than that of developing from scratch.

2.2 Information Islands

Unfortunately, software developers' anticipation of available reusable components does not always match real repository systems. Empirical studies on the use of high-functionality computing systems (reuse repository systems being typical examples of them) have found there are four levels of users' knowledge about a computing system (Fig. 1) [12]. Because users of reuse repository systems are software developers, we will substitute "software developers" for "users" in the following analysis.

In Fig. 1, ovals represent levels of a software developer's knowledge of a reuse repository system, and the rectangle represents the actual repository, labeled L4. L1 represents those components that are well known, easily employed, and regularly reused by a developer. L1 corresponds to the reuse-by-memory mode. L2 contains components known vaguely and reused only occasionally by a developer; they often require further confirmation when they are reused. L2 corresponds to the reuse-by-recall mode. L3 represents what developers believe, based on their experience, exists in the repository system. L3 corresponds to the reuse-by-anticipation mode.

Many components fall in the area of (L4 - L3), which means their existence is not known to the software developer. Consequently, there is little possibility for the developer to reuse them because people generally cannot ask for what they

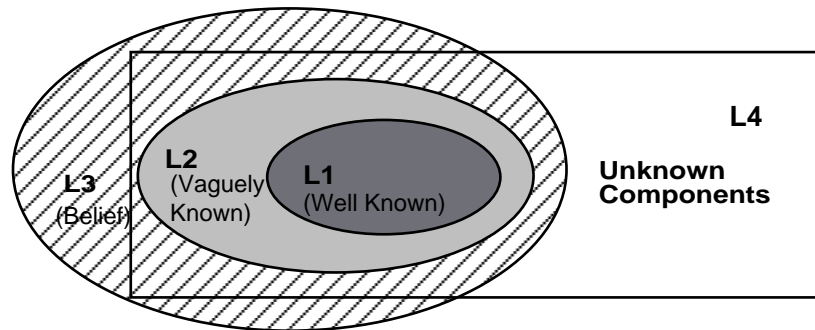


Fig. 1. Different levels of users' knowledge about a system

do not know [15]. Components in (L4 - L3) thus become *information islands* [8], inaccessible to software developers without appropriate tools.

Many reports about reuse experience in industrial software companies illustrate this inhibiting factor of reuse. Devanbu et al. [6] report that developers, unaware of reusable components, repeatedly re-implement the same function—in one case, this occurred ten times. This kind of behavior is also observed as typical among the four companies investigated by Fichman and Kemerer [10]. From the experience of promoting reuse in their organization, Rosenbaum and DuCastel conclude that making components known to developers is a key factor for successful reuse [33].

Reuse repository systems, most of which employ information access mechanisms only and operate in the style of “if you ask, I tell you,” provide little help for software developers to explore those information islands.

2.3 Reuse Utility

Human beings try to be utility-maximizers in the decision-making process [30]; software developers are no exception. Reuse utility is the ratio of reuse value to reuse cost. Although there is considerable cost involved in setting up the reuse repository, from the perspective of software developers, their perception of reuse cost consists of the part associated with the reuse process—cost of the location, comprehension, and modification of reusable components.

Reduction of location cost leads to the increase of reuse utility, which in turn leads to the selection of reuse approach. Location cost can be further broken down into the following items:

- (1) The effort needed to learn about the components, or at least the existence of them so that developers will initiate the reuse process.
- (2) The cost associated with switching back and forth between development environments and reuse repository systems. This switching causes the loss of working memory and the disruption of workflow [27].

- (3) The cost of specifying a reuse query based on the current task. Developers must be able to specify their needs in the way that is understood by a reuse repository system [13].
- (4) The cost of executing the retrieval process to find what is needed.

Automating the execution of the retrieval process only is not enough; reducing the costs of items (1), (2) and (3) should be given equal, if not more, consideration.

3 Active Information Delivery

In contrast to the conventional information access mechanism, in which users explicitly specify their information needs to computer systems, which in turn return retrieval results, the active delivery mechanism operates in the style that information is presented to users without being given explicit specifications of information needs. Active delivery systems that just throw a piece of decontextualized information at users, for example, Microsoft Office's *Tip of the Day*, are of little use because they ignore the working context. To improve the usefulness of delivered information so it can be utilized by users in accomplishing their tasks, relevance of the information to the task at hand or to the current working context must be taken into consideration. This context-sensitive delivery requires that active delivery systems have a certain understanding of what users are doing.

3.1 Active Information Delivery in Reuse Repository Systems

Equipping reuse repository systems with active information delivery not only makes it possible for software developers to reuse formerly unknown components, but also supports the seamless transition from development activities to reuse activities to reduce the cost of reuse.

A Bridge to Information Islands. Not knowing the existence of reusable components residing on information islands prohibits reuse from taking place. In contrast with passive reuse repository systems—systems employing query-based information access only, active reuse repository systems are able to compare the current development task with reusable components and proactively present those that are relevant to developers to increase the opportunity of reuse.

Well-Informed Decision Making. Studies on the human decision-making process have shown that the presence of problem-solving alternatives affects the final decision dramatically [30]. The presence of actively delivered reusable components reminds software developers of the alternative development approach—reuse—other than their current approach of developing from scratch. It prompts software developers to make well-informed decisions after giving due consideration to reuse.

Reduction of Reuse Cost. Active reuse repository systems reduce the cost of reuse by streamlining the transition from developing activities to component-locating activities. Software developers can access reusable components without switching their working contexts. This is less disruptive to their workflow compared to the use of passive repository systems because the latter involves more working memory lost. As software developers switch from development to reuse, their working memory (whose capacity is very limited and holds about 7 slots) of the development activities decays with a half-life of 15 seconds [27]. Therefore, the longer they spend on component-locating, the more working memory gets lost. With the support of active reuse repository systems, software developers do not need to specify their reuse queries explicitly and do not need to execute the searching process. All of these factors contribute to the reduction of reuse cost and the increase of reuse utility so that reuse can be put in a more favored situation.

3.2 Capturing the Task of Software Developers

For an active reuse repository system to deliver components relevant to the task in which a software developer is currently engaged, it must be able to capture to a certain extent what the task is.

Software development is a process of progressive transformation of requirements into a program. Inasmuch as software developers use computers, it is possible for reuse systems, if integrated with the development environment, to capture the task of software developers from their partially constructed programs, even though the reuse systems may not necessarily fully understand the task.

A program has three aspects: concept, code, and constraint. The concept of a program is its functional purpose or goal; the code is the embodiment of the concept; and the constraint regulates the environment in which the program runs. This characterization is similar to the 3C model of Tracz [36], who uses concept, content, and context to describe a reusable component.

A program includes not only its code part. Software development is essentially a cooperative process among many developers; therefore, programs must include both formal information for their executability and informal information for their readability by peer developers [35]. Informal information includes structural indentation, comments, and identifier names. Comments and identifier names are important beacons for the understanding of programs because they reveal the concepts of programs [2]. The use of comments and/or meaningful identifier names to index and retrieve software components has been explored [7, 9].

Modern programming languages such as *Java* enforce this self-explaining feature of programs further by introducing the concept of doc comments. A doc comment begins with `/**` and continues until `*/`. It immediately precedes the declaration of a module that is either a class or a method. The contents of doc comments describe the functionality of the following module. Doc comments are utilized by the *javadoc* program to create online documentation from *Java*

source codes. Most *Java* programmers therefore do not need to write separate, extra documents for their programs.

The constraint of a program is manifested by its signature. A signature is the type expression of a module that defines its syntactical interface. A signature of a function or a method specifies what types of inputs it takes and what types of outputs it produces. The signature of a class includes its data definition part and the collection of signatures of its methods. For a reusable component to be integrated, its signature should be compatible with the program to be developed.

Combining the concepts revealed through comments, as well as constraints revealed through signatures, it is highly possible to find components that can be reused in the current development task, if they show high relevance in concepts and high compatibility in constraints. Fortunately, in current development practices and environments, comments and signature definitions come sequentially before the code. This gives active repository systems a chance to deliver reusable components before the implementation of codes, after the systems have captured the comments and signatures and used them to locate relevant components automatically.

4 Implementation of an Active Reuse Repository System

A prototype of an active reuse repository system, *CodeBroker*, has been implemented. It supports *Java* programmers in reusing components during programming.

4.1 System Architecture

The architecture of *CodeBroker* is shown in Fig. 2. It consists of three software agents: *Listener*, *Fetcher*, and *Presenter*. A software agent is a software entity that functions autonomously in response to the changes in its running environment without requiring human guidance or intervention [3]. In *CodeBroker*, the *Listener* agent extracts and formulates reuse queries by monitoring the software developer's interaction with the program editor—*Emacs*. Those queries are then passed to *Fetcher*, which retrieves the matching components from the reuse repository. Reusable components retrieved by *Fetcher* are passed to *Presenter*, which uses user profiles to filter out unwanted components and delivers the filtered result in the *Reusable Components Info-display (RCI-display)*.

The reuse repository in *CodeBroker* is created by its indexing program, *CodeIndexer*, which extracts and indexes functional descriptions and signatures from the online documentation generated by running *javadoc* over *Java* source code.

4.2 Retrieval Mechanism

An effective retrieval mechanism is essential in any reuse repository system. *CodeBroker* uses the combination of latent semantic analysis (LSA) and signature matching (SM) as the retrieval mechanism. LSA is used to compute the

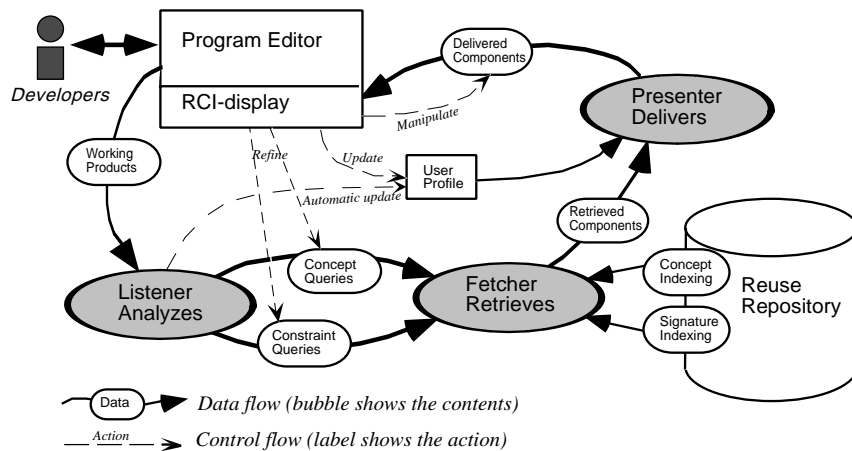


Fig. 2. The system architecture of *CodeBroker*

concept similarity existing between the concepts of the program under development and the textual documents of reusable components in the repository. SM is used to determine the *constraint compatibility* existing between the signature of the program under development and those of components in the repository.

Latent Semantic Analysis. LSA is a technology based on free-text indexing. Free-text indexing suffers from the concept-based retrieval problem; that is, if developers use terms different from those used in the descriptions of components, they cannot find what they want because free-text indexing does not take the semantics into consideration. By constructing a large semantic space of terms to capture the overall pattern of their associative relationship, LSA facilitates concept-based retrieval. The indexing process of LSA starts with creating a semantic space with a large corpus of training documents in a specific domain—we use the *Java* language specification, *Java* API documents, and *Linux* manuals as training documents to acquire a level of knowledge similar to what a *Java* programmer most likely has. It first creates a large term-by-document matrix in which entries are normalized scores of the term frequency in a given document (high-frequency words are removed). The 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 best value of k still remains an open question. 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 relationship among terms by

ignoring the noises accompanying most automatic thesaurus construction based simply on the 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, and so is a query. The similarity of a query and a reusable component is thus determined by the Euclidean distance of the two vectors. A reusable component matches a query if their similarity value is above a certain threshold. Compared to traditional free-text indexing techniques, LSA can improve retrieval effectiveness by 30% in some cases [5].

Signature Matching. SM is the process of determining the compatibility of two components in terms of their signatures [37]. It is an indexing and retrieval mechanism based on constraints. The basic form of a signature of a method is:

`Signature: InTypeExp->OutTypeExp`

where `InTypeExp` and `OutTypeExp` are type expressions resulted 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.

The above definition of SM is very restrictive because it misses components whose signatures do not exactly match but are similar enough to be reusable after slight modification. Partial signature matching relaxes the definition of structural conformance of types: A type is considered as conformant to its more generalized form or its more specialized form. For procedural types, if there is a path from type T1 to type T2 in the type lattice, T1 is a generalized form of T2, and T2 is a specialized form of T1. 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 T1 is a superclass of T2, T1 is a generalized form of T2, and T2 is a specialized form of T1.

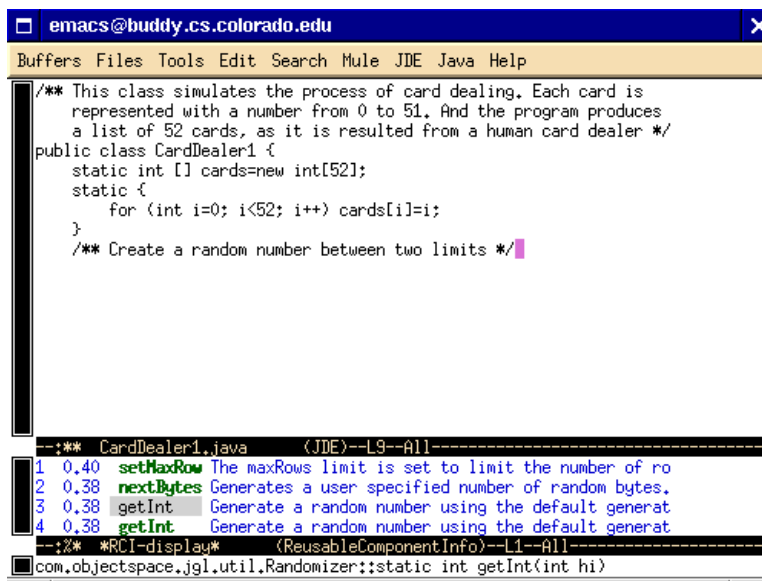
The constraint compatibility value between two signatures is the production of the conformance value between their types. The type conformance value is 1.0 if two types are in structural conformance according to the definition of the programming language. It drops a certain percentage (the current system uses 5% which will be adjusted as more usability experience is gained) if one type conversion is needed, or there is an immediate inheritance relationship between them, and so forth. The constraint compatibility value is 1.0 if two signatures exactly match.

4.3 Listener

The *Listener* agent runs continuously in the background of *Emacs* to monitor the input of software developers. Its goal is to capture the task software developers have at hand and construct reuse queries on behalf of them. Reuse queries are extracted from doc comments and signatures.

Whenever a software developer finishes the definition of a doc comment, *Listener* automatically extracts the contents, and creates a concept query that reflects the concept aspect of the program to be implemented. HTML markup tags included in the doc comments are stripped, as well as tagged information such as @author, @version, etc.

Figure 3 shows an example. A software developer wants to generate a random number between two integers. Before he or she implements it (i.e., writes the code part of the program), the task is indicated in the doc comment. As soon as the comment is written (where the cursor is placed), *Listener* automatically extracts the contents: **Create a random number between two limits**. This is used as a reuse query to be passed to *Fetcher* and *Presenter*, which will present, in the *RCI-display* (the lower part of the editor) those components whose functional description matches this query based on LSA.



```
emacs@buddy.cs.colorado.edu
Buffers Files Tools Edit Search Mule JDE Java Help

/** This class simulates the process of card dealing. Each card is
    represented with a number from 0 to 51. And the program produces
    a list of 52 cards, as it is resulted from a human card dealer */
public class CardDealer1 {
    static int [] cards=new int[52];
    static {
        for (int i=0; i<52; i++) cards[i]=i;
    }
    /** Create a random number between two limits */
}

---:** CardDealer1.java (JDE)--L9--All-----
1 0.40 setMaxRow The maxRows limit is set to limit the number of ro
2 0.38 nextBytes Generates a user specified number of random bytes.
3 0.38 getInt Generate a random number using the default generat
4 0.38 getInt Generate a random number using the default generat
---:** *RCI-display* (ReusableComponentInfo)--L1--All-----
com.objectspace.jgl.util.Randomizer::static int getInt(int hi)
```

Fig. 3. Reusable component delivery based on comments

Concept similarity is often not enough for components to be reused because they also need to satisfy the type compatibility. Incompatible types of reusable components inflict much difficulty in the modification process. For instance, in

Fig. 3, although component 3, the signature of which is shown in the message buffer (the last line of the window), could be modified to achieve the task, it is desirable to find a component that can be immediately integrated without modification.

Type compatibility constraints are manifested in the signature of a module. As the software developer proceeds to declare the signature, it is extracted by *Listener*, and a constraint query is created out of it. As Fig. 4 shows, when the developer types the left bracket { (just before the cursor), *Listener* is able to determine it as the end of a module signature definition. *Listener* thus creates a constraint query: `int x int -> int`. Figure 4 shows the result after the query is processed. Notice that the first component in the *RCI-display* in Fig. 4 has exactly the same signature—shown in the second line of the pop-up window—as the one extracted from the editor, and therefore can be reused immediately.

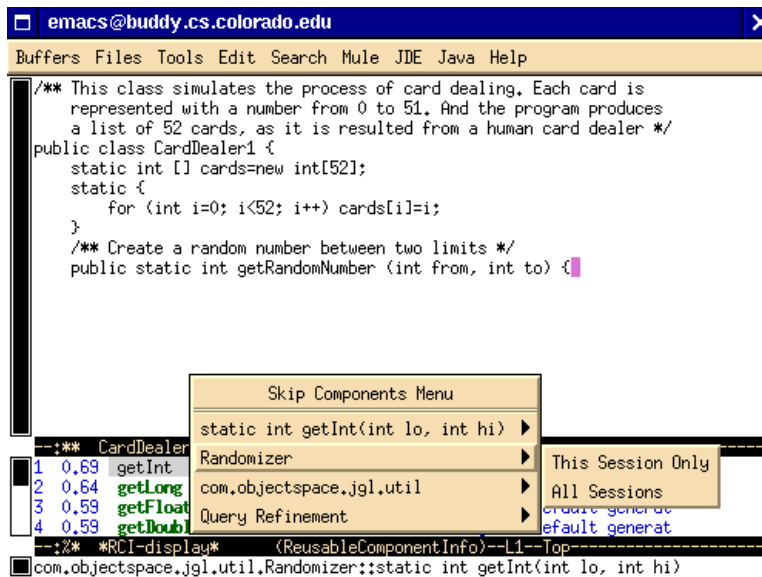


Fig. 4. Reusable component delivery based on both comments and signatures

4.4 Fetcher

The *Fetcher* agent performs the retrieval process. When *Listener* passes a concept query, *Fetcher* computes, using LSA, the similarity value from each component in the repository to the query, and returns those components whose similarity value ranks in the top 20. The number 20 is the threshold value used by *Fetcher* to determine what components are regarded as relevant, and can be customized by software developers. If software developers are not satisfied with

the delivery based on comments only and proceed to declare the signature of the module, the SM part of *Fetcher* is invoked to rearrange the previously delivered components by moving those signature-compatible ones into higher ranks. The new combined similarity value is determined using the formula

$$\text{Similarity} = \text{ConceptSimilarity} * w1 + \text{ConstraintCompatibility} * w2$$

where $w1 + w2 = 1$, and the default values for $w1$ and $w2$ are 0.5. Their values can be adjusted by software developers to reflect their own perspectives on the importance of concept similarity and constraint compatibility accordingly.

Figures 3 and 4 show the difference when the signature is taken into consideration. Component 1 in Fig. 4 was component 4 in Fig. 3, and the top three components in Fig. 3 do not even show up in the visible part of *RCI-display* in Fig. 4 because they all have incompatible signatures.

4.5 Presenter

The retrieved components are then shown to software developers by the agent *Presenter* in *RCI-display* in decreasing order of similarity value. Each component is accompanied with its rank of similarity, similarity value, name, and a short description. Developers who are interested in a particular component can launch, by a mouse click, an external HTML rendering program to go to the corresponding place of the full *Java* documents.

The goal of active delivery in *CodeBroker* is meant to inform software developers of those components that fall into L3 (reuse-by-anticipation) and the area of (L4 - L3) (information islands) in Fig. 1. Therefore, delivery of components from L2 (reuse-by-recall) and L1 (reuse-by-memory), especially L1, might be of little use, with the risk of making the unknown, really needed components less salient. *Presenter* uses a user profile for each software developer to adapt the components retrieved by *Fetcher* to the user's knowledge level of the repository, to ensure that those components already known to the user are not delivered.

A user profile is a file that lists all components known to a software developer. Each item on the list could be a package, a class, or a method. A package or a class indicates that all components from either of them should not be delivered; a method indicates that the method component only should not be delivered. The user profile can be updated by software developers through interaction with *Presenter*. A right mouse click on the component delivered by *Presenter* brings the **Skip Components Menu**, as shown in Fig. 4. A software developer can select the **All Sessions** command, which will update his or her profile so that the component or components from that class or that package will not be delivered again in later sessions.

The **Skip Components Menu** also allows programmers to instruct the *Presenter* to remove, in this session only, the component, or the class and package it belongs to, by selecting the **This Session Only** command. This is meant to temporarily remove those components that are apparently not relevant to the current task in order to make the needed component easier to find.

5 Related Work

This work is closely related to research on software reuse repository systems, as well as to active information systems that employ active information delivery mechanisms.

5.1 Software Reuse Repository Systems

Free-text indexing-based reuse repository systems take the textual description of components as the indexing surrogates. The descriptions may come from the accompanying documents [24], or be extracted from the comments and names from source code [7, 9]. Reuse queries are also written in natural language. The greatest advantage of this approach is its low cost in both setting up the repository and posing a query. However, this approach does not support concept-based retrieval.

Faceted classification is proposed by Prieto-Diaz to complement the incompleteness and ambiguity of natural language documents [29]. Reusable components are described with multiple facets. For each facet, there is a set of controlled terms. A thesaurus list accompanies each term so that reusers can use any word from the thesaurus list to refer to the same term. Although this approach is designed to improve the retrieval effectiveness, experiments have shown it does not perform better than the free-text based approach [17, 25].

AI-based repository systems use knowledge bases to simulate the human process of locating reusable components in order to support concept-based retrieval. Typical examples include *LaSSIE* [6], *AIRS* [28], and *CodeFinder* [18]. *LaSSIE* and *CodeFinder* use frames and *AIRS* uses facets to represent components. A semantic network is constructed by experts to link components based on their semantic relationship. The bottleneck to this approach is the difficulty in constructing the knowledge base, especially when the repository becomes large.

Although most reuse repository systems use the conceptual information of a component, the constraints, that is, the signatures, of components can also be used to index and retrieve components. Rittri first proposed the use of signatures to retrieve components in functional programming languages [32]. His work is extended in [37], which gives a general framework for signature matching in functional programming languages.

CodeBroker is most similar to those systems adopting free-text indexing. It is also similar to the AI-based systems because the semantic space created by LSA could be regarded as a simulation of human understanding of the semantic relationship among words. Whereas semantic networks used in AI-based systems are instilled with human knowledge, semantic spaces of LSA are trained with a very large corpus of documents. After being trained on about 2,000 pages of English texts, LSA has scored as well as average test-takers on the synonym portion of TOEFL [21]. *CodeBroker* also extends signature matching onto object-oriented programming languages. In terms of retrieval mechanisms, *CodeBroker* is unique because it combines both the concept and constraint of programs in order to improve the retrieval effectiveness, whereas all other systems use only one aspect.

Finally, the most distinctive feature of *CodeBroker* is its active delivery mechanism, which does not exist in any of above systems.

5.2 Active Information Systems

Reuse repository systems are a subset of information systems that help users find the information needed to accomplish their work from a huge information space [12]. Many information systems have utilized the active information delivery mechanism to facilitate such information use.

The most simple example of active information systems is Microsoft Office's *Tip of the Day*. When a user starts an application of Microsoft Office, a tip is given on how to operate the application. More sophisticated active information systems exploit the shared workspace between working environments and information systems to provide context-sensitive information. *Activists*, which monitors a user's use of *Emacs* and suggests better commands to accomplish the same task, is a context-sensitive and active help system [14]. *LispCritic* uses program transformation rules to recognize a less ideal code segment, and delivers a syntactical equivalent, but more efficient, solution [11].

Active delivery is also heavily utilized in the design of autonomous interface agents for WWW information exploration. *Letizia*, based on analyzing the current web page browsed by users and their past WWW browsing activities, suggests new web pages that might be of interest for their next reading [22]. *Remembrance Agent* utilizes the active delivery mechanism to inform users of those documents from their email archives and personal notes that are relevant to the document they are writing in *Emacs* [31].

CodeBroker is similar to those systems in terms of using current work products as retrieval cues to information spaces, and building a bridge to information islands with active delivery mechanisms.

6 Summary

Most existing reuse repository systems postulate that software developers know when to initiate a reuse process, although systematic analysis of reuse failures has indicated that no attempt to reuse is the biggest barrier to reuse [16]. When deployment of such repository systems fail, many blame the managerial issues, or the NIH (not invented here) syndrome, and call for education to improve the acceptance of reuse. Managerial commitment and education are indeed important for the success of reuse, but we feel it equally important to design reuse repository systems that are oriented toward software developers and are integrated seamlessly into their current working environments. Ready access to reusable components from their current working environments makes reuse appealing directly to software developers. As we have witnessed from the relative success of so-called ad hoc, individual-based reuse-by-memory, active reuse repository systems can extend the memory of software developers by presenting relevant reusable components right into their working environments.

Ongoing work on the development of *CodeBroker* aims to expand the signature matching mechanism into the class level with more relaxed matching criteria. An evaluation of *CodeBroker* will also be performed to gain better understanding of the difficulties encountered by software developers when they adopt reuse into their development activities, as well as to determine to which extent an active reuse repository system such as *CodeBroker* is able to help.

References

- [1] Basili, V., Briand, L., Melo, W.: How reuse influences productivity in object-oriented systems. *Comm. of the ACM*, 39(10):104–116, 1996.
- [2] Biggerstaff, T.J., Mitbander, B.G., Webster, D.E.: Program understanding and the concept assignment problem. *Comm. of the ACM*, 37(5):72–83, 1994.
- [3] Bradshaw, J.M.: *Software Agents*. AAAI Press, Menlo Park, CA USA, 1997.
- [4] Brooks, F.P.: *The Mythical Man-Month: Essays on Software Engineering*. Addison-Wesley, Reading, MA USA, 20th anniversary ed., 1995.
- [5] Deerwester, S., Dumais, S.T., Furnas, G.W., Landauer, T.K., Harshman, R.: Indexing by latent semantic analysis. *Journal of the American Society for Information Science*, 41(6):391–407, 1990.
- [6] Devanbu, P., Brachman, R.J., Selfridge, P.G., Ballard, B.W.: LaSSIE: A knowledge-based software information system. *Comm. of the ACM*, 34(5):34–49, 1991.
- [7] DiFelice, P., Fonzi, G.: How to write comments suitable for automatic software indexing. *Journal of Systems and Software*, 42:17–28, 1998.
- [8] Engelbart, D.C.: Knowledge-domain interoperability and an open hyperdocument system. In *Proc. of Computer Supported Cooperative Work '90*, 143–156, New York, NY USA, 1990.
- [9] Etzkorn, L.H., Davis, C.G.: Automatically identifying reusable OO legacy code. *IEEE Computer*, 30(10):66–71, 1997.
- [10] Fichman, R.G., Kemerer, C.E.: Object technology and reuse: Lessons from early adopters. *IEEE Software*, 14(10):47–59, 1997.
- [11] Fischer, G.: A critic for Lisp. In *Proc. of the 10th International Joint Conference on Artificial Intelligence*, 177–184, Los Altos, CA USA, 1987.
- [12] Fischer, G.: User modeling in human-computer interaction. *User Modeling and User-Adapted Interaction*, (to appear)
- [13] Fischer, G., Henninger, S., Redmiles, D.: Cognitive tools for locating and comprehending software objects for reuse. In *Proc. of the 13th International Conference on Software Engineering*, 318–328, Austin, TX USA, 1991.
- [14] Fischer, G., Lemke, A.C., Schwab, T.: Knowledge-based help systems. In *Proc. of Human Factors in Computing Systems '85*, 161–167, San Francisco, CA USA, 1985.
- [15] Fischer, G., Reeves, B.N.: Beyond intelligent interfaces: Exploring, analyzing and creating success models of cooperative problem solving. In Baecker, R., Grudin, J., Buxton, W., Greenberg, S., (eds.): *Readings in Human-Computer Interaction: Toward the Year 2000*, 822–831, Morgan Kaufmann Publishers, San Francisco, CA USA, 2nd ed., 1995.
- [16] Frakes, W.B., Fox, C.J.: Quality improvement using a software reuse failure modes models. *IEEE Transactions on Software Engineering*, 22(4):274–279, 1996.

- [17] Frakes, W.B., Pole, T.P.: An empirical study of representation methods for reusable software components. *IEEE Transactions on Software Engineering*, 20(8):617–630, 1994.
- [18] Henninger, S.: An evolutionary approach to constructing effective software reuse repositories. *ACM Transactions on Software Engineering and Methodology*, 6(2):111–140, 1997.
- [19] Isoda, S.: Experiences of a software reuse project. *Journal of Systems and Software*, 30:171–186, 1995.
- [20] Krueger, C.W.: Software reuse. *ACM Computing Surveys*, 24(2):131–183, 1992.
- [21] Landauer, T.K., Dumais, S.T.: A solution to Plato's problem: The latent semantic analysis theory of acquisition, induction and representation of knowledge. *Psychological Review*, 104(2):211–240, 1997.
- [22] Lieberman, H.: Autonomous interface agents. In *Proc. of Human Factors in Computing Systems '97*, 67–74, Atlanta, GA USA, 1997.
- [23] Lim, W.C.: Effects of reuse on quality, productivity and economics. *IEEE Software*, 11(5):23–29, 1994.
- [24] Maarek, Y.S., Berry, D.M., Kaiser, G.E.: An information retrieval approach for automatically constructing software libraries. *IEEE Transactions on Software Engineering*, 17(8):800–813, 1991.
- [25] Mili, H., Ah-Ki, E., Grodin, R., Mcheick, H.: Another nail to the coffin of faceted controlled-vocabulary component classification and retrieval. In *Proc. of Symposium on Software Reuse '97*, 89–98, Boston, MA USA, 1997.
- [26] Mili, H., Mili, F., Mili, A.: Reusing software: Issues and research directions. *IEEE Transactions on Software Engineering*, 21(6):528–562, 1995.
- [27] Norman, D.A.: Cognitive engineering. In Norman, D.A., Draper, S.W., (eds.): *User centered system design: New perspective on human-computer interaction*, 31–61, Lawrence Erlbaum Associates, Hillsdale, NJ USA, 1986
- [28] Ostertag, E., Hendler, J., Prieto-Diaz, R., Braun, C.: Computing similarity in a reuse library system: An AI-based approach. *ACM Transactions on Software Engineering and Methodology*, 1(3):205–228, 1992.
- [29] Prieto-Diaz, R.: Implementing faceted classification for software reuse. *Comm. of the ACM*, 34(5):88–97, 1991.
- [30] Reisberg, D.: *Cognition*. W. W. Norton & Company, New York, NY, 1997.
- [31] Rhodes, B.J., Starner, T.: Remembrance agent: A continuously running automated information retrieval system. In *Proc. of the 1st International Conference on the Practical Application of Intelligent Agents and Multi Agent Technology*, 487–495, London, UK, 1996.
- [32] Rittri, M.: Using types as search keys in function libraries. *Journal of Functional Programming*, 1(1):71–89, 1989.
- [33] Rosenbaum, S., DuCastel B.: Managing software reuse—an experience report. In *Proc. of 17th International Conference on Software Engineering*, 105–111, Seattle, WA USA, 1995.
- [34] Simon, H.A.: *The Sciences of the Artificial*. The MIT Press, Cambridge, MA USA, 3rd ed., 1996.
- [35] Soloway, E., Ehrlich, K.: Empirical studies of programming knowledge. *IEEE Transactions on Software Engineering*, SE-10(5):595–609, 1984.
- [36] Tracz, W.: The 3 cons of software reuse. In *Proc. of the 3rd Workshop on Institutionalizing Software Reuse*, Syracuse, NY USA, 1990.
- [37] Zaremski, A.M., Wing, J.M.: Signature matching: A tool for using software libraries. *ACM Transactions on Software Engineering and Methodology*, 4(2):146–170, 1995.