Proc. of 8th International Symposium on Foundations of Software Engineering, San Diego, CA, pp 60-68, 2000

Integrating Active Information Delivery and Reuse Repository Systems

Yunwen Ye[1,2]
[1]Software Research Associates, Inc.
3-12 Yotsuya, Shinjuku
Tokyo 160-0004, Japan
+81-3-3357-9361
yunwen@cs.colorado.edu

Gerhard Fischer[2]
[2]Department of Computer Science
University of Colorado at Boulder
Boulder, CO80303, USA
+1-303-492-1502
gerhard@cs.colorado.edu

Brent Reeves[3]
[3]TwinBear Research
Boulder, CO80303, USA
+1-303-499-2666
brent@twinbear.com

ABSTRACT
Although software reuse can improve both the quality and productivity of software development, it will not do so until software developers stop believing that it is not worth their effort to find a component matching their current problem. In addition, if the developers do not anticipate the existence of a given component, they will not even make an effort to find it in the first place.

Even the most sophisticated and powerful reuse repositories will not be effective if developers don't anticipate a certain component exists, or don't deem it worthwhile to seek for it. We argue that this crucial barrier to reuse is overcome by integrating *active information delivery,* which presents information without explicit queries from the user, and reuse repository systems. A prototype system, *CodeBroker,* illustrates this integration and raises several issues related to software reuse.

Categories and Subject Descriptors
D.2.13 [**Software Engineering**]: Reusable Software—*Reusable libraries, Reuse models*; D.2.2 [**Software Engineering**]: Design Tools and Techniques—*Computer-aided software engineering, User interfaces*; H.5.2 [**Information Interfaces and Presentation**] User Interfaces—*Interactive styles, User-centered design*

General Terms
Design, Human Factors

Keywords
Software reuse, active information delivery, active reuse repository, developer-centered CASE, latent semantic analysis, signature matching

1. INTRODUCTION
A wide gap exists between the constantly increasing demands of complex software systems and the capability of the software industry to deliver quality software systems in a timely and cost-effective manner. Software reuse, a development method of using existing reusable software components to create new systems, has been shown through empirical studies to improve both the quality and productivity of software development [1]. Software reuse also increases the evolvability of software systems because complex systems evolve faster when they are built from stable subsystems [35]. However, before developers can take advantage of reuse, they must be able to locate reusable components easily and quickly.

This paper focuses on the issues surrounding reusable component location. We apply cognitive engineering [31] to build a model of the reuse process. Based on this cognitive model and our past research on the effective use of large information spaces [15], we identify barriers to component location. Current reuse repository systems do not sufficiently support the whole range of reusable component location activities, especially in the phase of forming reuse intentions. We argue that reuse repository systems should be designed from a developer-centered perspective for reuse to be widely embraced by developers. A prototype of a developer-centered reuse repository system, *CodeBroker*, is described.

CodeBroker runs as an active process in a software development environment. It infers developers' needs for reusable components by monitoring their interaction with the development environment. Potentially reusable components are located and delivered to the developer in a non-intrusive manner.

2. BARRIERS TO COMPONENT REUSE
Current reuse repository systems are designed to support the development-with-reuse process model. Development-with-reuse treats the reuse process as a separate process to be incorporated into development practices. This is a methodology-centered perspective. It requires that developers change their current practice to adopt reuse. Against this view, we argue that even though reuse may be a goal for the company, it is not a primary goal of developers; it is only one of many means they use to accomplish their development tasks. Only when developers perceive reuse as having immediate benefit, will reuse be readily embraced [22].

2.1 A Cognitive Model of the Reuse Process
Software reuse is primarily a cognitive activity that starts with a goal in the mind of a developer. To achieve such goals,

developers have to translate internal intentions into a series of physical actions constrained by a reuse repository system [31]. Figure 1 illustrates the actions necessary for the reuse process to succeed. First, developers must consciously decide to use the repository, we call this forming a reuse intention. Second, they must formulate their intention as a query in the vocabulary of the reuse repository. Third, when the repository returns matching components, developers must choose the right one to integrate into their work. The following sections analyze in detail the challenges of each step.

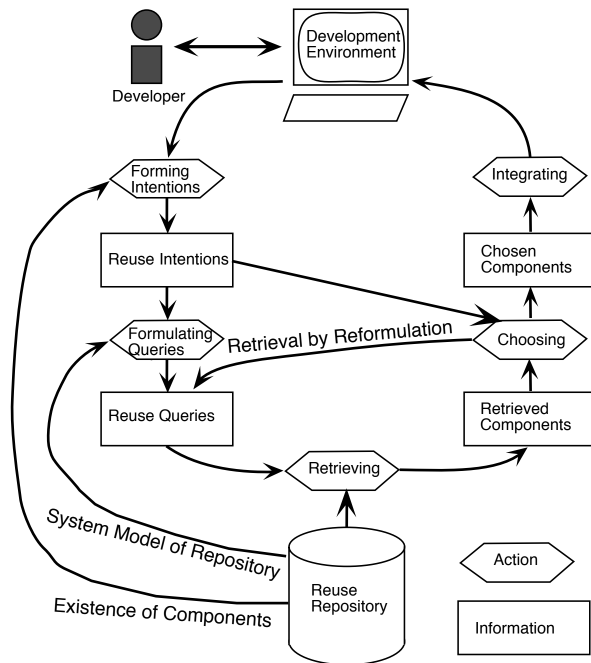

Figure 1: A Cognitive Model of the Reuse Process

2.1.1 Vocabulary Learning: A Prerequisite for Forming Reuse Intentions

Reusable components help developers think at higher levels of abstraction. Like the introduction of a new word into the English language that increases our ability to reason and communicate, reusable components increase the ability of the developer to create and interpret designs [26]. However, developers must learn the syntax and the semantics of the new vocabulary to take advantage of reuse repositories and to be able to form and express their reuse intentions. *Vocabulary learning* is a major part of the cognitive barrier to reuse [3].

2.1.2 Conceptual Gap in Formulating Reuse Queries

Formulating reuse queries refers to transforming internal reuse intentions into explicit, external reuse queries. Reuse intentions, derived from development activities, are conceptualized in the situation model [25] that is related to the application task to be solved and to the concerns of the developer. A situation model is the mental model a developer has of his environment. A system model is the "actual" model of a computer system. For a component to be retrieved, these intentions need to be mapped from the user's situation model onto the system model, namely the repository system [13]. Without enough knowledge about the system model of reuse repository systems, developers cannot formulate reuse queries appropriately. This *conceptual gap* between situation and system model is another cognitive barrier to reuse.

2.1.3 Effective Retrieval Mechanisms

Executing the retrieval process involves finding the components that match given reuse queries. An effective retrieval mechanism—including a representation schema for the purpose of indexing and matching criteria between a query and a component—is essential. A complete and precise representation can make the matching more precise and retrieval more effective. However, because the same representation is also used by developers to specify their reuse queries, the format of representation is limited by the developers' willingness to formulate long and precise queries [30].

2.1.4 Retrieval by Reformulation

Retrieved components only match reuse queries. It is difficult for most reusers to create a well-defined query on their first attempt. *Retrieval by reformulation* is the process through which queries are incrementally improved until they express the query that matches their reuse intentions [10, 37].

2.1.5 Integration Requires Comprehension

In order to integrate a reusable component into their program, developers must first comprehend it. In some cases, the reusable component needs to be modified and this requires that the developer understand how the component is implemented. Our previous research has demonstrated that explanations linking the concept embodied in the component to its implementation are useful [17].

2.2 Information Islands in Reuse Repository

There are three modes for a developer to use a reuse repository: *reuse-by-memory*, *reuse-by-recall,* and *reuse-by-anticipation*.

In the reuse-by-memory mode, while designing a new program, developers may notice similarities between the new program and reusable components whose names and functionality are well known; therefore, they can directly reuse those components with no difficulty. Because they remember the components, they can find them without much effort.

In the reuse-by-recall mode, developers remember that a component exists, but don't remember exactly which component it was. They know the repository contains relevant material and know it might be worth searching.

In the reuse-by-anticipation mode, developers formulate reuse intentions based on their anticipation of the existence of certain reusable components. Though 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 relevant components.

Unfortunately, developers' anticipation of available reusable components does not always match real reuse repository systems. Empirical studies on the use of high-functionality computing systems (e.g. reuse repository systems) have discovered that there are four levels (L) of user knowledge (Figure 2) [15].

In Figure 2, ovals represent levels of knowledge of an information space, and the rectangle represents the actual information space, labeled L4. L1 represents those elements (reusable components) that are well known, easily employed, and regularly used (reused) by a developer. L1 corresponds to the reuse-by-memory mode. L2 contains components known vaguely and reused only occasionally by a developer; and they often require further confirmation before being reused. L2 corresponds to the reuse-by-recall mode. L3 represents what developers believe about the repository. L3 corresponds to the reuse-by-anticipation mode.

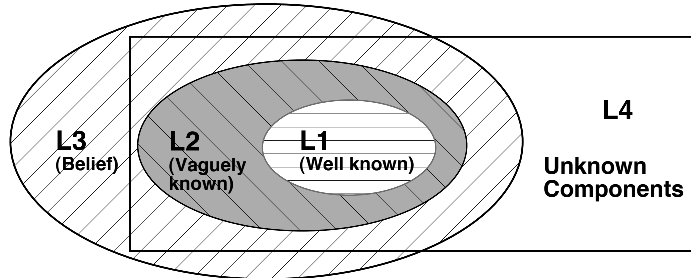

Figure 2: Different Levels of Developers' Knowledge about Reuse Repository

Many components exist in the area of (L4 – L3), and their existence is not known to developers. Consequently, there is little possibility for developers to reuse them because people do not ask for what they do not know [18]. Components in (L4 – L3) thus become *information islands* [7], inaccessible to developers without appropriate tools. As reuse repositories grow, this area also grows. Repositories are not static—it is expected that they will evolve over time, and this will increase the size of (L4 – L3).

Many reports about reuse experience of industrial software companies illustrate this inhibiting factor of reuse. Devanbu et al. [5] report that because developers are unaware of reusable components, they 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 in [9]. From the experience of promoting reuse, Rosenbaum and DuCastel [34] conclude that making components known to developers is a key factor for successful reuse. Developers will reuse those components repeatedly once they have reused them once [24].

2.3 Cognitive Biases against Reuse

Human beings often try to be utility-maximizers in the decision-making process [33] and developers are no exception. Reuse utility is the ratio of reuse value to reuse cost. All actions identified in §2.1 have associated costs. Additionally, reuse costs also include the reallocation of working memory and the disruption of workflow caused by switching back and forth between development environments and reuse repository systems.

There is no objective estimation formula for reuse value and reuse cost. The estimation made by developers during their development activities is quite subjective and suffers from cognitive biases against reuse.

Once problem solvers discover a strategy that "gets the job done," they are less likely to discover new strategies until they are completely stuck [33]. Even today, for most developers, building programs from scratch is the proven strategy. This partially explains the observed phenomenon of "developer machoism"—developers have a tendency to chronically underestimate how difficult a programming task is and overestimate the costs of reuse [21].

Another known phenomenon in the decision-making process of human beings is *loss aversion*—the tendency to be far more sensitive to potential loss than to potential gain [33]. Starting a reuse process requires a mental switch. The demand on working memory and time is immediate, while the potential gain is unclear. Reducing the perceived loss of reuse is thus very important.

3. ACTIVE INFORMATION DELIVERY IN REUSE REPOSITORY SYSTEMS

Current reuse repository systems are passive in the sense that they support only query-based information access. Developers must formulate reuse queries explicitly. However, because reuse queries are generated from the development activities and most software development activities are carried out with computer-aided software engineering tools, it is possible for reuse repository systems to play a more active role in supporting reuse.

3.1 Active Information Delivery

In active information delivery systems [2, 15, 16], the relevant information is delivered at the right moment for a problem solver to consider. In contrast to the query-based information access mechanism, active information delivery systems present information to developers without having been asked for it explicitly.

Simon [35] has pointed out that cognitive activities are determined by the environment in which they take place. The environment includes information present in the workspace (development environment in our case), and information present in 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. Based on this analysis, reuse-by-memory and reuse-by-recall incorporate information of reusable components in the memory of developers triggered by cues in the development environment. Active information delivery plays a similar role to augment the memory of developers. It uses the cues present in the development environment to predict the needs for reusable components. Based on the prediction, it locates and delivers potentially reusable components.

Active information delivery builds a bridge to information islands in a reuse repository by bringing unknown components to the attention of developers. It reduces reuse costs by eliminating the

steps of query formulation as well as the context switch between the development environment and a reuse repository system [38]. Reuse repository systems equipped with active information delivery mechanisms are called active reuse repository systems.

A critical challenge for active delivery systems is the contextualization of new information to the task at hand. Active systems ("push components") which 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. Our previous research on incorporating active information delivery into domain-oriented design environments [16] identified that the delivered information must be relevant both to the user's task at hand, and the background knowledge of the user himself [12].

3.2 Relevance to the Task at Hand

Delivering information relevant to the task at hand requires that one use the information in the development environment to anticipate the user's information needs.

A software 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 is the environment in which it runs. This characterization is similar to the 3C model of Tracz [36], who uses concept, content and context to describe a reusable component.

Important concepts of a program are often contained in informal information structures. Software development is essentially a cooperative process among many developers. Programs include both formal information for executability and informal information for readability by peer developers. 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 important concepts of programs [2, 9, 29].

One important constraint of a program is its type compatibility, which is manifested in its signature. A signature is the type expression of a program, and defines its syntactical interface. For a reusable component to be easily integrated, its signature should be compatible with the environment into which it is going to be incorporated. Other constraints of a program can be its implementation language, its performance, its quality and so forth.

Combining the concept revealed through comments, as well as constraints revealed through signatures, it is possible to find a component that can be reused in the current development task. If the component shows high similarity in concept and high compatibility in constraint the likelihood of a component matching the developer's task is high. Fortunately, in current development environments, comments and signature definitions come sequentially before the code. This gives active repository systems a chance to deliver reusable components before the developer has implemented the code. As soon as the signature is defined or the comment is written, the active repository system can begin to search for matching components while the developer proceeds to write the code.

3.3 Relevance to the User

A piece of information that is helpful to one developer may be distracting to others. Even for the same developer, as his knowledge level about the information space of a reuse repository grows, his needs for information on reusable components will also change. User profiles, which represent the users' varying preferences and knowledge levels about the system, can be used in reuse repository systems to adapt the system behavior to each user (developer) [12]. User profiles can be used to keep track of the components that have been reused in the past. Depending on the context, these components can be omitted from certain lists. The system can assume a certain level of familiarity with those components and deduce that the user would have reused them if he thought they were relevant. Thus, the user's previous interaction with the system can serve as a filter for future interaction.

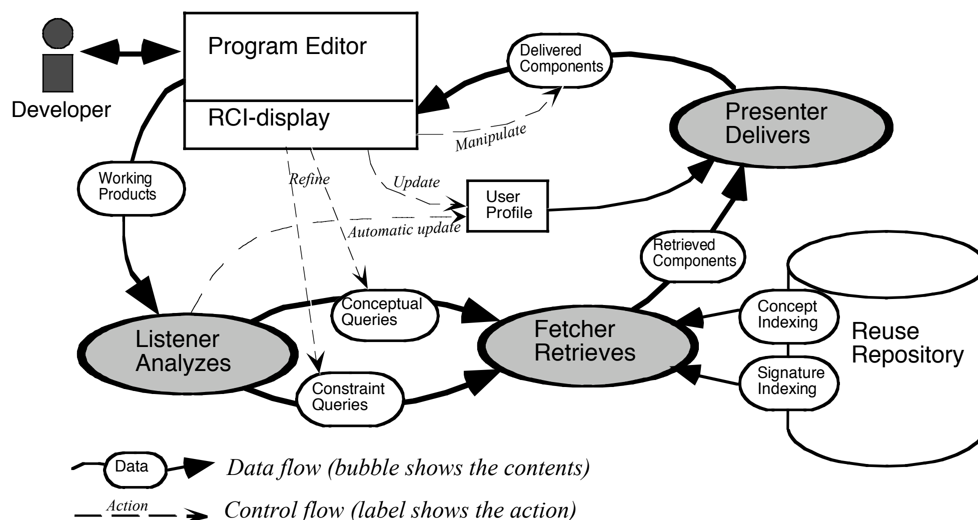

Figure 3: The Architecture of CodeBroker

4. CODEBROKER: AN ACTIVE REUSE REPOSITORY SYSTEM

CodeBroker is a prototype of an active reuse repository system. It runs autonomously in the background of a program editor (*Emacs* in our case) and utilizes active information delivery to:

- show the developers components that have a high probability of being reused in their current development task,
- reduce the overall cost of the component location process by eliminating the step of explicit query formulation, and
- eliminate the need for the developer to switch contexts between development environment and reuse repository system.

The architecture of *CodeBroker* is shown in Figure 3. 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 [2]. In *CodeBroker,* the *Listener* agent extracts and formulates reuse queries by monitoring developers' interaction with the editor. 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 the user profile to filter out unwanted components and delivers them in the *Reusable Components Info-display (RCI-display)* placed beneath the editor window.

The reuse repository in *CodeBroker* is created by the indexing program, *CodeIndexer*, which extracts and indexes functional descriptions and signatures from the online documentation generated by running *javadoc* over Java source code. Each component is indexed in two ways: concept indexing and signature indexing. Concept indexing indexes components based on their functional descriptions in text. Signature indexing creates a signature representation for each component. The current repository of *CodeBroker* consists of the Java 1.1.7 Core API library and JGL 3.0 (Java General Library from ObjectSpace Inc.), and there are about 10,000 components that are either a class or a method.

4.1 Listener

The *Listener* agent runs continuously in the background of *Emacs* to monitor the input of developers. Its goal is to extract and formulate reuse queries based on the cues present in the development environment. Reuse queries are automatically extracted from comments and signatures. Because of the automatic formulation of reuse queries, developers are not prevented from reusing the component even though they may not have learned that vocabulary yet.

4.1.1 Extracting Conceptual Queries

The specific information being extracted by *Listener* is the *document comments* in Java programs. Document comments are special programming conventions introduced by Java. They begin with `/**` and continue until the next `*/`, and immediately precede the declaration of a module—either a class or a method. Document comments are meant to describe the functionality of the following module. Whenever a developer finishes the definition of a document comment, *Listener* automatically extracts the content, and creates a query based on it, which we call conceptual query.

Figure 4 shows an example in which a developer wants to generate a random number between two integers, and before she implements it (i.e. writes the code part of the program) the developer indicates her task in the document comment. As soon as the comment is written, *Listener* extracts the content: "`Create a random number between two limits.`" This is used as a conceptual query to be passed to *Fetcher* and *Presenter,* which will use the *RCI-display* (the lower part of the editor) to present components whose functional description matches this query. The exact retrieval mechanism is described later.

4.1.2 Extracting Constraint Queries

Conceptual similarity is often not enough for components to be reused because they also need to match in terms of constraints. For instance, in Figure 4, although component 3, the signature of which is shown in the message buffer (the last line of the

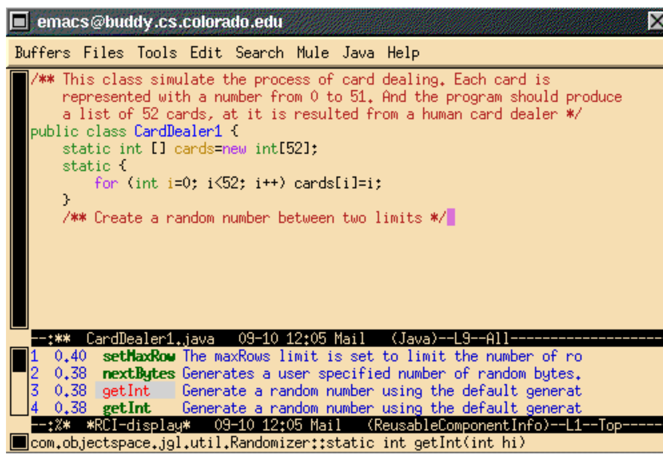

Figure 4: Reusable Components Delivered Based on Comments

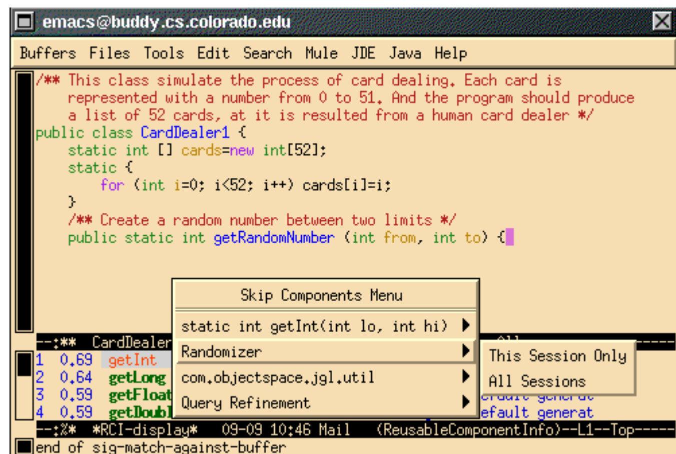

Figure 5: Reusable Components Delivered Based on Both Comments and Signatures

window), could be modified or wrapped to achieve the task, it would be better to find a component that can be immediately integrated without modification.

Type compatibility constraints are manifested in the signature of a program. As the developer proceeds to declare the signature, it is extracted by *Listener;* and a query is created from it. Queries in the form of a signature are called constraint queries. As Figure 5 shows, when the developer types the left bracket `{` (just before the cursor) *Listener* is able to recognize it as the end of a module signature definition. *Listener* thus creates a constraint query: `int x int -> int`. Figure 5 shows the result after the constraint query is processed. Notice that the first component in the *RCI-display* in Figure 5 has exactly the same signature (shown in the second line of the pop-up window) as the one extracted from the editor, and can be reused without further modification.

4.2 Fetcher

The retrieval mechanism used by *Fetcher* is the combination of *Latent Semantic Analysis* (LSA) [27] and *Signature Matching* (SM) [39]. LSA is used to compute the conceptual similarity between conceptual queries and functional descriptions of reusable components. SM is used to determine the constraint compatibility between constraint queries and signatures of reusable components.

4.2.1 Latent Semantic Analysis

LSA is a technology based on free-text indexing. Free-text indexing suffers from the concept-based retrieval problem. 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 and fills in the conceptual gap in formulating reuse queries.

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, and the value of k often ranges from 40 to 400 while 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 three reduced matrices. In this new matrix, each row represents the position of each term in the semantic space. Terms are re-represented in the newly created semantic space. The reduction of singular values is important because it captures only the major, overall pattern of associative relationships among terms by ignoring the noises accompanying most automatic thesaurus construction simply based on co-occurrence statistics of terms.

After the semantic space is created, each reusable component is represented as a vector in the semantic space based on terms contained, 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. LSA makes it possible for developers to retrieve reusable components on the basis of conceptual content.

4.2.2 Signature Matching

Signature matching is the process of determining the compatibility of two components in terms of their signatures [39]. It is an indexing and retrieval mechanism based on type constraints. The basic form of a signature of a method is:

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

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

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

the signature is

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

Two signatures

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

match if and only if `InTypeExp1` is in structural conformance with `InTypeExp2`, and `OutTypeExp1` is in structural conformance with `OutTypeExp2`. Two type expressions are structurally conformant if they are formed by applying the same type constructor to structurally conformant types.

The above definition of signature matching is very restrictive because it misses some components whose signature does not exactly match, but which are in practice similar enough to be reusable after slight modification or with wrappers added.

Partial signature matching relaxes the definition of structural conformance of types: A type is considered as conforming to its more generalized form or more specialized form. For procedural types, if there is a path from `T1` to `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 subclass of `T2`, `T1` is a specialized form of `T2`, and `T2`, is a generalized 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% and will be adjusted as we gain usability experience) if one type conversion is needed, or there is an immediate inheritance relationship between them, and so forth. The signature compatibility value is 1.0 if two signatures exactly match.

4.3 Presenter

The retrieved reusable components are shown to developers by the *Presenter* agent in the *RCI-display* in decreasing order of similarity value. Each component is accompanied with its rank of similarity, its similarity value, its name, and a short description (Figure 4). Developers who are interested in a particular

component can launch, by a mouse click, an external browsing interface, which could be any HTML-text browser, to go to the corresponding place of the full Java documents. When signature is taken into consideration, the similarity value is the average value of the conceptual similarity value returned by LSA and the constraint compatibility value returned by SM process (Figure 5).

Presenter uses user profiles to adapt the retrieved components to the knowledge level of each developer. A user profile is a file listing all components known to the developer. Each item of the list could be a package, a class, or a method. A package indicates none of its classes and methods should be delivered; a class indicates none of its methods should be delivered; a method indicates only that method should not be delivered.

User profiles are automatically updated by the *Listener* agent. If the *Listener* agent has observed that the user has reused a component for several times (currently, 3), it assumes that there is no needs to deliver this component any more because if it is reusable the developer would have reused it by memory or by recall, and adds this component to her user profile.

The user profile can also be edited by developers manually, or updated through the interaction with *CodeBroker*. A right mouse click on the component delivered by *Presenter* brings the `Skip Components Menu`, as shown in Figure 5. Developers can select the `All Sessions` command, which will update their profiles, and that component or components from that class or that package will not be delivered again in later sessions.

The `Skip Components Menu` allows developers to advise 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 remove components that are not relevant to the current task in order to make the needed components easier to find.

The last command in the `Skip Components Menu` would bring developers to a query refinement interface where they can modify the automatically formulated query and launch a new location process.

5. RELATED RESEARCH ON REUSE REPOSITORY SYSTEMS

The importance of reuse repository systems in supporting reuse has been recognized from the very beginning of reuse research. Although some systems support the choosing process, most of them try to automate the retrieval process.

5.1 Reuse Repository Systems Supporting Retrieval

Several different retrieval mechanisms have been proposed. Based on the representation schema of components, they can be divided into text-based, structured representation-based, and formal method-based approaches.

The text-based approach uses textual descriptions as surrogates to represent reusable components, and adopts information retrieval techniques to define the similarity between reuse queries, also represented in text, and reusable components. Textual descriptions can be drawn from accompanying documents, such as in *GURU* [28], or they can be extracted from comments and/or identifier names in reusable components [4, 8].

Although the text-based approach is easy both for setting up a reuse repository and for reusers to formulate reuse queries, it does not provide support in shortening the conceptual gap in query formulation. Reusers must use the same terms used in the description of reusable components to locate what they need.

Structured representation-based approaches try to fill in this conceptual gap by employing a knowledge base or conceptual distance graph to mimic the method a human being would employ in locating reusable components. Multi-faceted classification schemas [6] use multiple facets to represent a reusable component and a conceptual distance graph to reflect the semantic relationships among terms describing reusable components. Both *CodeFinder* [23] and *LaSSIE* [5] represent reusable components as frames. *CodeFinder* organizes those frames into an associated network and uses spreading activation to find reusable components. Frames in *LaSSIE* are organized into hierarchical, taxonomic categories. It is difficult to construct such systems because they need to formalize expert knowledge that is difficult to elicit, and they are difficult to scale.

Formal method-based approaches use either signatures [39], or formal specifications [40] to represent components. Signature-based approaches are easy to use, but suffer from the impreciseness of representation; formal specification approaches are time consuming and cognitively challenging for reusers.

CodeBroker is most similar to those systems adopting text-based approach. Text-based approach has shown comparable retrieval effectiveness to other more retrieval mechanisms [20]. Compared to traditional text indexing and retrieval mechanism, LSA can improve retrieval effectiveness by 30% in some cases [27]. We expect the retrieval effectiveness of *CodeBroker* can be further improved by combining both the concept-based, or functional description-based, indexing and constraint-based, or signature-based, indexing. Few reuse repository systems have ever tried to index and retrieve reusable components based on the combination of two aspects. This speculation needs to be substantiated with experiments in our future research.

5.2 Reuse Repository Systems Supporting Choosing

In our past research, we have noticed that requirements for reusable components are by nature ill defined, and that locating reusable components is not a one-time effort. We have added retrieval by reformulation to the reuse repository so reusers can incrementally develop queries based on the results of a previous search. *Helgon* [14] and *CodeFinder* [13] have demonstrated how retrieval by reformulation can help reusers choose reusable components by incrementally improving their queries.

Choosing the right component can also be supported by showing examples of how those components are used in similar situations. *Explainer* illustrates how the component is reused in a related example [17].

CodeWeb helps developers choose components from different repositories by providing different views. It uses name matching and information retrieval-based similarity matching to identify similar reusable components [29].

6. DISCUSSION

Software development is difficult because of the irreducible essential difficulties brought by the complexity, conformity,

changeability and invisibility of software systems [3]. Software reuse is an effort to reduce these difficulties, but it introduces new accidental difficulties of operating the reuse repository. *CodeBroker* mitigates the new difficulties by taking care of a part of the routine work. Developers can focus their attention on the essential aspects of software development instead of worrying about a query interface to a reuse repository. However, *CodeBroker* does not address all the difficulties identified in §2.1. While it helps developers form reuse intentions and reduces the conceptual gap of reuse query formulation by employing new retrieval mechanism, its support in choosing and integrating is very limited. *CodeBroker* is not meant to be the only approach to the access of reusable components; it complements query-based and browsing mechanisms.

Some developers may find this kind of active delivery too intrusive. Empirical studies are needed to evaluate the effectiveness of *CodeBroker* in supporting reuse. If developers write comments that do not reflect the functionality, or do not write comments at all, the usefulness of *CodeBroker* will be greatly diminished.

Even though the location process of reusable components may become less demanding with the support of systems such as *CodeBroker*, it is still not clear whether software companies are willing to pay the costs associated with setting up, evolving and sustaining reuse repositories. Traditionally, there are two distinctive roles: producers and consumers of reusable components [30]. Producers of components apply domain analysis and other techniques to identify and develop reusable components, which are then consumed (reused) by developers. This dichotomous perspective on reuse is in line with the traditional viewpoint of software systems: developers who only produce the system, and users who only use the system. The success stories of open-source systems (OSS) [32] such as Linux, which treat users as co-developers, fundamentally challenge the assumed dichotomy of producers and consumers of reuse repository systems, and provide an alternative way to the production (setting up, evolving and sustaining) of reuse repositories [11].

One important factor in the success of OSS is to get users involved as co-developers. We must provide immediate benefits to the developers so that they will help build better repository systems. Therefore, lowering the barriers to the access of reusable components not only contributes to the consumption of, but also possibly leads to the production of, reuse repositories. In the future, we will customize *CodeBroker* to support *Jun for Java*—an open source reuse repository of 3D graphics developed and distributed by Software Research Associates, Inc. (http://www.srainc.com/Jun4Java/index-e.html), and analyze its role in the consumption and incremental production of reuse repositories with the OSS approach.

Although *CodeBroker* is designed to promote reuse in the phase of coding, the underlying principles are equally applicable to higher levels of software development activities. For example, if developers use modeling tools such as *Rational Rose* to create a conceptual design of a software system, they need to specify the functionality and signature for each class. An active reuse repository system can utilize that information to actively deliver potentially reusable components.

7. SUMMARY

Successful reuse requires developers know what to reuse and when to reuse. Despite the fact that empirical studies have found that lack of reuse intentions (i.e. “no attempts to reuse”) is the most important impediment to the adoption of reuse [19], current reuse repository systems offer little support. We propose the concept of active reuse repository systems that employ active information delivery mechanisms. We have demonstrated its feasibility with the development of a prototype, *CodeBroker*. It utilizes information present in the development environment to aid in the retrieval of potentially reusable components. Active reuse repository systems can increase the utility and effectiveness of reuse repositories because they:

- find those components whose existence is not even anticipated by developers;
- reduce reuse costs by eliminating the reuse query formulation step;
- alleviate the cognitive biases against reuse by highlighting relevant reusable components to developers; and
- augment developer’s memory by delivering potentially reusable components that trigger the adoption of reuse.

8. ACKNOWLEDGEMENTS

The authors would like to thank Jonathan Ostwald, Eric Scharff, and Rogerio de Paula who provided much feedback to the conceptual framework and system discussed in this paper. This research is supported by Software Research Associates, Inc. Tokyo, and PFU Limited, Tokyo.

9. REFERENCES

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

[2] Bradshaw, J.M., *An Introduction to Software Agents*, in *Software Agents*, J.M. Bradshaw, Editor. 1997, AAAI Press: Menlo Park, CA. p. 1-46.

[3] Brooks, F.P.J., *The Mythical Man-Month: Essays on Software Engineering*. 20th Anniversary ed., Addison-Wesley (1995).

[4] Burton, B., et al., "The Reusable Software Library," *IEEE Software*, **4**(4), 25-33 (1987).

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

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

[7] Engelbart, D.C., "Knowledge-Domain Interoperability and an Open Hyperdocument System," *Proc. Computer Supported Cooperative Work 1990*, 143-156, (1990), New York, NY.

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

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

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

[11] Fischer, G., "Beyond 'Couch Potatoes': From Consumers to Designers," *Proc. 1998 Asia-Pacific Computer and Human Interaction*, 2-9, (1998), Kanagawa, Japan.

[12] Fischer, G., "User Modeling: The Long and Winding Road," *Proc. User Modeling 1999*, 349-355, (1999), Banff, Canada.

[13] Fischer, G., S. Henninger, and D. Redmiles, "Cognitive Tools for Locating and Comprehending Software Objects for Reuse," *Proc. 13th ICSE*, 318-328, (1991), Austin, TX USA.

[14] Fischer, G., A.C. Lemke, and C. Rathke, "From Design to Redesign," *Proc. 9th ICSE*, 369-376, (1987), Monterey, CA.

[15] Fischer, G., A.C. Lemke, and T. Schwab, "Knowledge-Based Help Systems," *Proc. Human Factors in Computing Systems 1985*, 161-167, (1985), San Francisco, CA.

[16] Fischer, G., et al., *Embedding Critics in Design Environments*, in *Readings in Intelligent User Interfaces*, M.T. Maybury and W. Wahlster, Editors. 1998, Morgan Kaufmann Publisher. p. 537-559.

[17] Fischer, G., et al., "Beyond Object-Oriented Development: Where Current Object-Oriented Approaches Fall Short," *Human-Computer Interaction, Special Issue on Object-Oriented Design*, **10**(1), 79-119 (1995).

[18] Fischer, G. and B.N. Reeves, *Beyond Intelligent Interfaces: Exploring, Analyzing and Creating Success Models of Cooperative Problem Solving*, in *Readings in Human-Computer Interaction: Toward the Year 2000*, R. Baecker, et al., Editors. 1995, Morgan Kaufmann Publishers: San Francisco, CA. p. 822-831.

[19] Frakes, W.B. and C.J. Fox, "Quality Improvement Using a Software Reuse Failure Modes Models," *IEEE Trans. Soft. Eng.*, **22**(4), 274-279 (1996).

[20] Frakes, W.B. and T.P. Pole, "An Empirical Study of Representation Methods for Reusable Software Components," *IEEE Trans. Soft. Eng.*, **20**(8), 617-630 (1994).

[21] Graham, I., "Reuse: A Key to Successful Migration," *Object Magazine*, **5**(6), 82-83 (1995).

[22] Grudin, J., "Groupware and Social Dynamics: Eight Challenges for Developers," *Commun. of the ACM*, **37**(1), 92-105 (1994).

[23] Henninger, S., "An Evolutionary Approach to Constructing Effective Software Reuse Repositories," *ACM Trans. Soft. Eng. Meth.*, **6**(2), 111-140 (1997).

[24] Isoda, S., "Experiences of a Software Reuse Project," *J. of Systems and Software*, **30**, 171-186 (1995).

[25] Kintsch, W., *Comprehension: A Paradigm for Cognition*. , Cambridge University Press: Cambridge, England (1998).

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

[27] Landauer, T.K. and S.T. Dumais, "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).

[28] Maarek, Y.S., D.M. Berry, and G.E. Kaiser, "An Information Retrieval Approach for Automatically Constructing Software Libraries," *IEEE Trans. Soft. Eng.*, **17**(8), 800-813 (1991).

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

[30] Mili, H., F. Mili, and A. Mili, "Reusing Software: Issues and Research Directions," *IEEE Trans. Soft. Eng.*, **21**(6), 528-562 (1995).

[31] Norman, D.A., *Cognitive Engineering*, in *User Centered System Design, New Perspectives on Human-Computer Interaction*, D.A. Norman and S.W. Draper, Editors. 1986, Lawrence Erlbaum Associates, Inc.: Hillsdale, NJ. p. 31-61.

[32] O'Reilly, T., "Lessons from Open-Source Software Development," *Commun. of the ACM*, **42**(4), 33-37 (1999).

[33] Reisberg, D., *Cognition*. , W. W. Norton & Company: New York, NY (1997).

[34] Rosenbaum, S. and B. DuCastel, "Managing Software Reuse--An Experience Report," *Proc. 17th ICSE*, 105-111, (1995), Seattle, Washington.

[35] Simon, H.A., *The Sciences of the Artificial*. Third ed., The MIT Press: Cambridge, MA (1996).

[36] Tracz, W., "The 3 Cons of Software Reuse," *Proc. 3rd Annual Workshop on Institutionalizing Software Reuse (WISR '90)*, (1990), Syracuse, NY.

[37] Williams, M.D., et al., "RABBIT: Cognitive Science in Interface Design," *Proc. 4th Annual Conference of the Cognitive Science Society*, 82-85, (1982), Ann Arbor, MI.

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

[39] Zaremski, A.M. and J.M. Wing, "Signature Matching: A Tool for Using Software Libraries," *ACM Trans. Soft. Eng. Meth.*, **4**(2), 146-170 (1995).

[40] Zaremski, A.M. and J.M. Wing, "Specification Matching of Software Components," *ACM Trans. Soft. Eng. Meth.*, **6**(4), 333-369 (1997).