An Active and Adaptive Reuse Repository System

Article · April 2001

DOI: 10.1109/HICSS.2001.927260 · Source: CiteSeer

1 author:

Yunwen Ye
University of Colorado Boulder
58 PUBLICATIONS **2,927** CITATIONS

An Active and Adaptive Reuse Repository System

Yunwen Ye
Software Engineering Laboratory, Software Research Associates, Inc.
3-12 Yotsuya, Shinjuku, Tokyo 160-0004, Japan
and
Department of Computer Science, CB430
University of Colorado at Boulder, Boulder, CO80309-0430, USA
Email: yunwen@cs.colorado.edu Tel:+1-303-492-8136

Abstract

Although software reuse repository systems have been an active research area for more than a decade, one important aspect has not been given enough attention: If software developers make no attempt to reuse, repository systems offer no help and reuse will not happen. Active information delivery, which presents information without being given explicit queries can motivate software developers to reuse. Critical challenges for active delivery systems are the contextualization of delivered information to the development task, and the adaptation of the information to each developer. This paper discusses how to address these challenges and demonstrates the feasibility of implementation through a prototype system.

1. Introduction

Software development is a process of progressive crystallization of software developers' knowledge into a system. Lack of needed knowledge is one of the major reasons that cause poor quality and productivity. With the advent of objected-oriented technology, reusable software components now become an indispensable part of the knowledge required for software development. Supporting easy access to needed external information, reusable components in particular, to complement the insufficient knowledge of software developers is thus critical to the improvement of the quality and productivity of software development [1].

Before software developers can reuse, they have to locate reusable components quickly and easily. Reusable component location is often supported by reuse repository systems. Current reuse repository systems do not sufficiently support the whole range of reusable component location activities, especially in the phase of forming reuse intentions (i.e. to make the conscious decision to reuse) [25]. Most current reuse repository systems are designed as standalone tools separated from development tools. Software developers have to initiate the reuse process on their own. However, many software developers fail to do so because they often do not anticipate correctly the existence of available reusable components [7].

To assist software developers in forming reuse intentions—the first step to the success of reuse, a reuse repository system needs to be integrated seamlessly with current development practice and tools. This integration can be realized with the *active information delivery* mechanism. Active information delivery presents information without being given explicit specification of information needs by users. The critical challenges of realizing active information delivery lie on how the system can capture user's information needs from their working environments, and how the system can adapt the delivered information to each user to improve the usefulness of the delivered information. This paper analyzes what kind of information available in a development environment can be used as cues indicating software developers' needs for reusable components, and how such information can be utilized to actively deliver relevant reusable components.

Reuse repository systems equipped with active information delivery are called *active reuse repository systems.* Furthermore, if the systems adapt the delivered information to each software developer, they become *active and adaptive reuse repository systems.* To illustrate the concept of active and adaptive reuse repository systems, a prototype system *CodeBroker* will be presented. *CodeBroker*

(1) helps software developers identify reuse opportunities by delivering reusable components unanticipated by them and yet relevant to their current development task, and
(2) reduces the overall cost of the component location process through eliminating the step of explicit query formulation and the needs to switch contexts between development environments and reuse repository systems.

2. Problems with Current Reuse Repository Systems

2.1. Development-with-Reuse

Most current reuse repository systems are designed to support the development-with-reuse process model [18]. This process model views reuse as a standalone process, independent of current development processes and tools, and postulates that software developers are always willing to reuse and have no difficulty in forming reuse intentions and formulating reuse queries. This viewpoint leads to the separation of reuse repository systems and software development tools. Whenever a development task arises, software developers must divert from their current processes to execute the reuse process [17]. If they fail to do so, reuse repository systems are of no use, and reuse will not happen. In fact, this is a very critical hurdle to the success of reuse; empirical studies have shown that "no attempt to reuse" is the most significant reuse failure mode [8].

2.2. No Attempt to Reuse

No attempt to reuse is caused by the existence of *information islands* in a reuse repository [24]. As illustrated in Fig.1, information islands contain components unanticipated by software developers, and they cannot be reused by software developers because people generally cannot ask for information they do not believe existing. Many reports about reuse experience in software companies illustrate the phenomenon of information islands. Devanbu et al. have reported that developers, unaware of reusable components, repeatedly re-implement the same function—in one case, ten times [3]. Similar reports can be found in [5, 20].

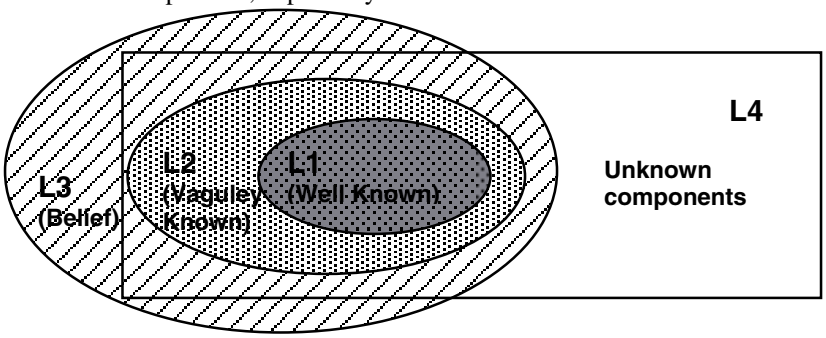
No attempt to reuse is also caused by the perceived low reuse utility—reuse utility is the ratio of reuse value to reuse cost. Due to cognitive biases against reuse, software developers tend to underestimate the reuse value and to overestimate the reuse cost [24]. This perceived low reuse utility prevents them from exploring those components falling in information islands and L3.

2.3. Reuse-within-Development

Development-with-reuse may work if all development activities can be planned a priori. However, in practice, software development is iterative and opportunistic. New development tasks arise all the time during the whole period of development; so does the reuse opportunities. Reuse can not be completely planned beforehand, it happens within the context and process of development [22].

In order to make reuse appealing and attractive to software developers, a perspective shift from development-with-reuse to reuse-within-development is needed. Putting software developers into the center of the design of reuse repository systems, and reuse into the development context as a whole, reuse-within-development views reuse as a supporting, not a replacing, method to the current practice of software developers. It requires the reuse process be smoothly melded into the current development processes, environments and tools so that there is no context switch between development and reuse.

Active reuse repository systems support reuse-within-development because they can be integrated with current



L4 is the set of components in a repository.
L1 is the set of components well known to developers, which can be easily reused.
L2 is the set of components vaguely known, whose functionality needs to be confirmed
L3 is the set of components that developers believe existing.
L4 – L3 is the area of information islands including components whose existence is unanticipated, and they can not be accessed by developers with the support of current reuse repository systems.

Figure 1: Information islands in a reuse repository

development environments and tools. They help software developers identify reuse opportunities whenever they arise. Since active reuse repository systems automatically create reuse queries and locate reusable components, software developers do not need to constantly switch back and forth between development environments and reuse repository systems. Reuse becomes an integral part, not an added-on activity, of the software development process.

3. Active and Adaptive Reuse Repository Systems

Critical challenges for the implementation of active information delivery systems are the contextualization of information to the task at hand and the adaptation to the different knowledge level of each developer. Active 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.

3.1. Contextualized Component Delivery

Delivering information relevant to the task at hand requires the system use the information in the development environment to predict the information needs of users. Relevance of information to the current task can be determined by either goal acquisition or similarity analysis.

The goal acquisition approach uses rules to specify the link from a series of actions, which is the condition part of the rule, to a task, which is the result part of the rule. When actions of a user match the condition, the system deems the user is performing the corresponding task and information about the task is delivered. Design critic systems often adopt this approach [6].

The similarity analysis approach examines the contextual information surrounding the current focus of users, and uses that information to predicate their information needs. Information from the repository that has high similarity to the contextual circumstance is then delivered. Systems like *Remembrance Agent* [19] and *Letizia* [12] fall into this category.

In the case of reusable component location, a goal acquisition approach will be similar to the recognition of a program plan [23]. It needs to recognize program plans from the program under development, and deliver reusable components that can be used to realize the recognized program plans. Recognition of program plans from complete programs are extremely difficult, let alone from partially constructed programs. Therefore, this approach is not suitable for active reuse repository systems.

A similarity analysis approach is to make use of the descriptive elements of programs and find components having similar descriptions. We adopt similarity analysis to find relevant reusable components.

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.

Important concepts of a program are often contained in its informal information structures. Software development is essentially a cooperative process among many software developers. Programs include both formal information for executability and informal information for readability by peer software developers. Informal information includes structural indentation, comments, and identifier names [23]. Comments and identifier names are important beacons for the understanding of programs [4, 14].

Modern programming languages such as *Java* enforce the inclusion of self-explaining informal information further by introducing the concept of *doc comments*. A doc comment begins with `/**` and continues until the next `*/`, and it immediately precedes the declaration of a module that is either a class or a method. Contents inside doc comments describe the functionality of the following module.

Constraints of a program are captured by its signature. A signature is the type expression of a program that defines its syntactical interface. For a reusable component to be easily integrated, its signature should be compatible with the environment to be incorporated into.

Relevance of reusable components to the current development task can be determined by the combination of *concept similarity* and *constraint compatibility*. Concept similarity is the similarity existing from the concept of the current task revealed through comments and identifiers to the concept revealed in the documents of reusable components. Constraint compatibility is the type compatibility existing from the signature of the program under development to those of reusable components.

3.2. Adapting to Each Developer

A piece of information helpful to one developer may be distracting to others. Even for the same developer, as his or her knowledge about the reuse repository grows, his or her needs for information will also change. In order to reduce the intrusiveness, which is the degree of developers' perception of being interrupted from their current focus, active information delivery systems should tune the delivered information to the needs of each user.

User profiles, representing the users' varying preferences and knowledge levels about the system, can be used by active information systems to adapt the system behavior to each user to improve the efficiency of communication between users and systems. User profiles can be explicitly modified by users or implicitly updated by the system. Modification by users requires the system be adaptable, which means users can customize the system behaviors to their own needs. Adaptive systems automatically update user profiles based on information observed or inferred from monitoring users' interactions with the system.

Adaptability and adaptivity complements with each other. Although adaptivity requires little effort from users, it needs a relatively long time to establish a reliable user profile. Adaptability gives users direct control over the delivery strategy, but it places extra work on users.

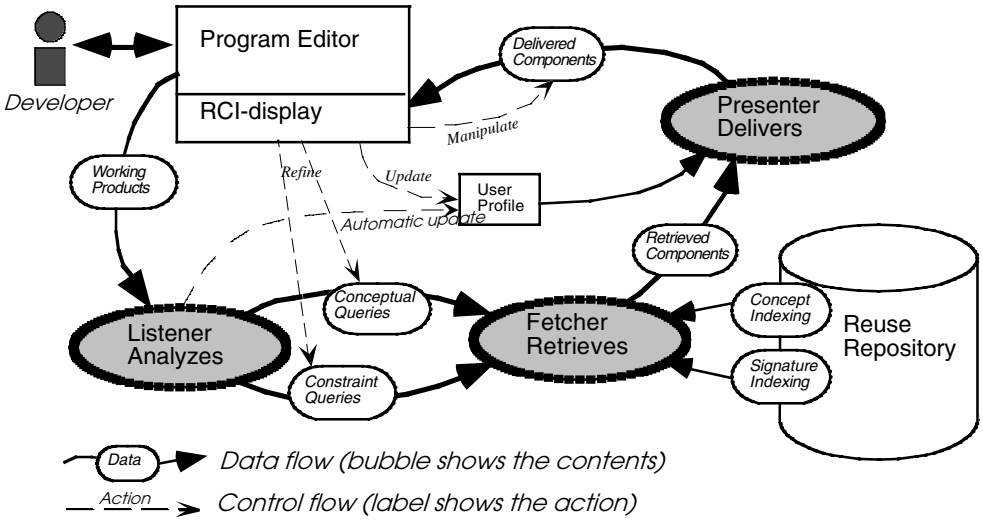


Figure 2: The architecture of CodeBroker

4. A Prototype: *CodeBroker*

We have developed an active and adaptive reuse repository system, *CodeBroker*, to assist *Java* developers in reusing classes and methods. Although the current repository consists of *Java* API library and JGL library (a *Java* General Library developed by ObjectSpace, Inc.) only, it can be easily extended to include reusable components locally developed or purchased from third parties. Its interface is integrated with the current development environment—*Emacs*. This integration makes the development environment a shared workspace of software developers and *CodeBroker*. Using software developers' working context as retrieval cues, *CodeBroker* automatically locates and delivers relevant reusable components right into the development environment.

The system 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. *Listener* and *Presenter* are interface agents that connect software developers to the backend information agent *Fetcher*. *Listener* captures software developers' needs for reusable components and formulates reuse queries autonomously. Upon receiving reuse queries, *Fetcher* retrieves relevant reusable components based on concept similarity and constraint compatibility. Retrieved components are presented by *Presenter* in the *RCI-display* after taking into consideration the difference of information needs of each developer. *RCI-display* is embedded in the development space; therefore, software developers do not need to leave their working environment to find needed reusable components.

4.1. Listener

As an interface agent, *Listener* runs continuously as a background process of the development environment. Whenever a software developer finishes a doc comment, *Listener* automatically extracts the contents and creates a concept query reflecting the concept aspect of the program under development.

Figure 3 shows an example of reusable component delivery. A software developer wants to generate a random number between two integers. Before he or she implements it (i.e., writes the code part), the task is

indicated in the doc comment. As soon as the comment is written (where the cursor is placed), *Listener* extracts the contents: `Create a random number between two limits`. This is used as a concept query to be passed to *Fetcher* and *Presenter*, which will present in the *RCI-display* those components whose functional descriptions match the query.

If the developer can not find the needed component immediately, he or she may proceed to declare the signature of the program. As Fig. 4 shows, when the developer types the left bracket { (just before the cursor), *Listener* recognizes it as the end of a signature definition. *Listener* thus creates a constraint query: `int x int -> int`. The *RCI-display* in Fig. 4 displays the result after the constraint query is processed. Notice that the first component 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. It can be reused immediately.

4.2. Fetcher

Fetcher is a backend information agent retrieving components from the repository that show high

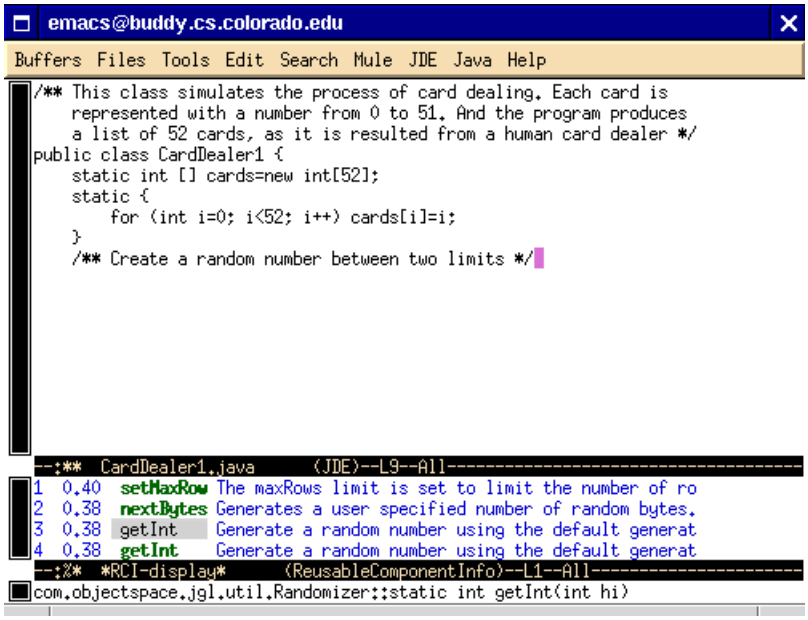


Figure 3: Component delivery based on concept queries

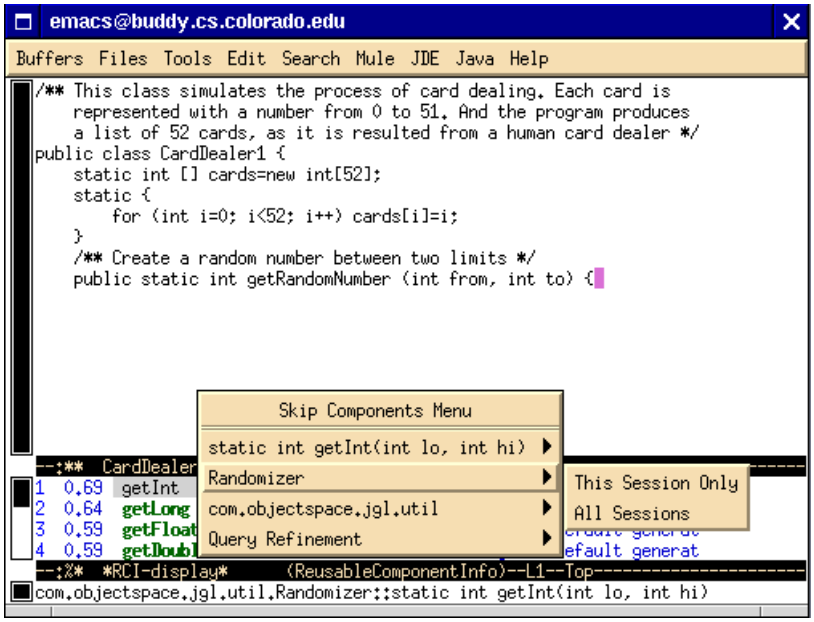


Figure 4: Component delivery based on concept queries and constraint queries

conceptual similarity and high constraint compatibility to the queries.

The conceptual similarity is determined by Latent Semantic Analysis (LSA) [11]. LSA is a technology based on free-text indexing [21]. The indexing process of LSA starts with creating a semantic space with a large corpus of training documents in a specific domain. It first creates a term by document matrix in which entries are normalized scores of the term frequency. 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 reduced to *k* dimension by eliminating small singular values; the value of *k* often ranges in 40-400. A new matrix, viewed as the semantic space of the domain, is reconstructed through the multiplication of the three reduced matrices. In this new matrix, each row represents the position of each term in the semantic space. The reduction of singular values is meant to capture only the major, overall pattern of associative relationships 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 and query are represented as vectors in the semantic space based on the terms contained. The similarity between a query and a reusable component is thus determined by the Euclidean distance of the two vectors. Comparing to traditional free-text indexing techniques, LSA makes it possible to retrieve by conceptual contents. LSA can improve up to 30% in terms of retrieval effectiveness in some cases [11].

The constraint compatibility is determined by Signature Matching (SM). SM is a process that computes the compatibility of two components in terms of their signatures [26]. It is an indexing and retrieval mechanism based on the constraints of programs. The basic form of a signature of a method is

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

where `InTypeExp` and `OutTypeExp` are type expressions resulted from applying Cartesian product constructor onto the input parameter types and output parameter types respectively. Two signatures

```
Sig1:InTypeExp1->OutTypeExp1

Sig2:InTypeExp2->OutTypeExp2
```

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

The constraint compatibility value between two signatures is the product of the conformance value existing among their types. The type conformance value is 1.0 if two types are in structural conformance according to the definition of the programming language; it drops a certain percentage if one type conversion is needed, or there is an immediate inheritance relationship existing between them, and so forth [24].

4.3. Presenter

The retrieved components are then shown to software developers in the *RCI-display* by the agent *Presenter* in decreasing order of similarity value. When conceptual similarity value only is available, the similarity value is equal to the conceptual similarity value returned by LSA; when both conceptual similarity value and constraint compatibility value are available, the similarity value is computed using the formula

$$ConceptSimilarity*w_1+ConstraintCompatibility*w_2$$

where $w_1+w_2=1$, and the default values are 0.5 respectively. Their values can be adjusted by software developers to reflect their own perspectives on the importance of concept similarity and constraint compatibility accordingly.

In the *RCI-display*, 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.

4.4. User Profiles in CodeBroker

The goal of active delivery is meant to inform software developers of those components that fall into L3 and the area of information islands (L4-L3) in Fig.1. Delivery of known components from L2 and L1, might be of little use, with the risk of making the unknown, really needed components less salient. If those known components can be reused in the current task, they would have been reused by software developers themselves.

Presenter uses a user profile for each software developer to adapt the components retrieved by *Fetcher* to the developer's knowledge level. User profiles in *CodeBroker* are files read in when the system is started. They contain two parts. The first part keeps the preference of the threshold value determining the relevance of reusable components, and the weights given to the concept similarity and constraint compatibility when they are combined. Reusable components whose similarity values are higher than the threshold are considered as relevant to the current task. Higher threshold value reduces the number of delivered reusable components, and improves the relevance to the task. Lower threshold value brings more selections.

The second part of a user profile is a list of the components known to the software developer. Each item on the list could be a package, a class, or a method. A package and a class indicate that no components from them should be delivered; a method indicates that the method component only should not be delivered.

User profiles are both adaptable and adaptive. Because of the large volume of reusable components and the constantly changing nature of reuse repository systems, asking software developers to maintain their user profiles is not realistic. Since the development environment is accessible to *CodeBroker*, the system can track what components are known to the software developer if it has observed that they have been reused. On the other hand, software developers may learn to reuse components from other venues, such as browsing, searching, reading documents, or discussion with peer developers. Allowing software developers to update their user profiles to reflect their newly acquired knowledge is also necessary.

4.4.1. Adaptability of User Profiles. User profiles can be updated by software developers through interactions with *CodeBroker*. A right mouse click on the component delivered in *RCI-display* 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 (the second line in the menu) or all components from that class (the third line) or that package (the forth line) will not be delivered again.

4.4.2. Adaptivity of User Profiles. *CodeBroker* continues to learn about the knowledge level of developers through monitoring their interactions with the system to increase the relevance of delivery. In addition to extract reuse queries, *Listener* also tracks what components from the repository are reused by a software developer. Based on the accumulated data, if *CodeBroker* finds that a software developer has reused a component more than the predefined threshold value, for example 3 times (which is adjustable by each developer), it will add that component into the developer's user profile and will not deliver it again.

4.5. Retrieval by Reformulation Interface

Locating reusable components is not the same as searching for data in a database where queries are well defined and completely articulated. Reuse queries and reusable components are not directly represented—a direct representation should be program codes; they are represented by surrogates such as textual descriptions and signatures. Those representations are partial and imprecise.

To complement the incompleteness of reuse queries, *CodeBroker* supports two forms of *retrieval by reformulation* [10]: direct manipulation and query refinement. After examining the retrieval results, initially delivered by *CodeBroker*, software developers can either refine the query to improve its completeness and preciseness, or directly manipulate the retrieval results by removing apparently irrelevant information.

Direct manipulation of the retrieval results is meant to incrementally create a shared understanding between the system and the task in which a developer is engaged. By invoking the command `This Session Only` of the pop-up menu `Skip Components Menu` associated with each

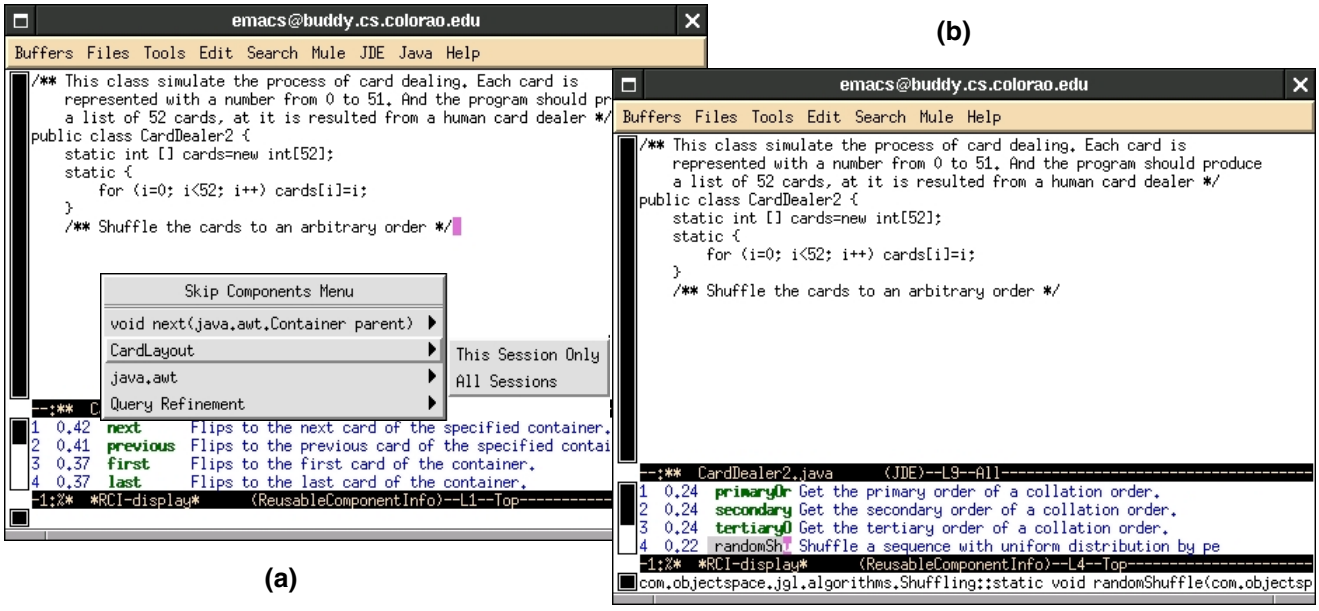


Figure 5: Direct manipulation

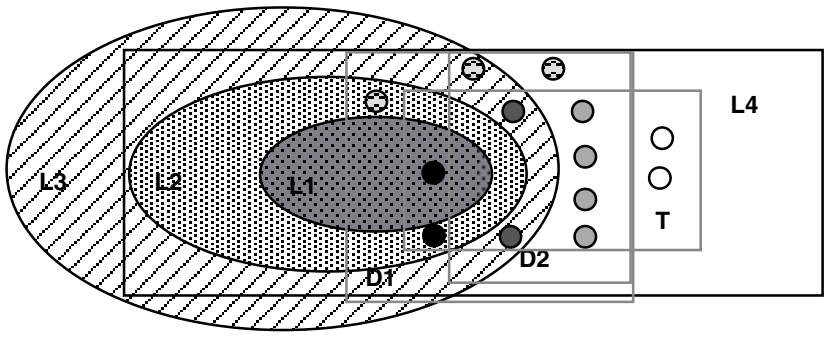
delivered component, software developers can temporarily remove those components not related to their current task and focus on examining those relevant. At the same time, the repository system also learns the scope of the development task, and improves incrementally the relevance of later deliveries by not displaying components having been explicitly removed by developers. For example, in Fig. 5(a), in response to the doc comment, *CodeBroker* delivers some components (No. 1 through No. 4) belonging to the class `java.awt.Cardlayout` (a GUI class) due to the shared keyword `card`. However, the current task of the software developer is not related to the class `Cardlayout`, so he or she can remove it through the direct manipulation interface. This manipulation brings the needed component `randomShuffle`, obscured previously, to the salient fourth place (Fig. 5(b)). Furthermore, no components from the class `java.awt.Cardlayout` will be delivered later in this development session.

Query refinement is invoked by clicking the `Query Refinement` command in the same pop-up menu, or directly typed in as an *Emacs* command. A buffer will appear for software developers to refine the automatically extracted reuse queries and start another round of component locating. Software developers can refine the conceptual query by choosing more appropriate terms, or modify the constraint query to make it less restrictive or more restrictive depending on the situation.

The retrieval by reformulation interface is a more comprehensive approach to improving the performance of the system than the relevance feedback technique in many information retrieval systems. Through the adjustment of terms used in a query by query expansion or other techniques, relevance feedback of information retrieval systems focuses mainly on the improvement of the retrieval system itself [2]. Instead, our focus is to improve the relevance of delivered components to the working context of software developers, not to the query per se. The direct manipulation tries to establish a shared understanding of the context between the reuse system and the software developer. It uses software developer's previous interactions with the system as a filter for later deliveries. Although it does not affect what *Fetcher* returns, it modifies what gets shown. The system also takes advantage of the fact that software components are organized into a hierarchy (packages, classes, and methods) according to their application domains to let software developers limit the retrieval to their interests. In the query refinement interface, software developers can describe specifically what packages or classes they are not interested in, or alternatively specify what parts of the repository are their current interests. This can limit the number of displayed components—not the retrieval results per se—to ease the task of choosing the needed component.

4.6. The Role of Each Agent

Figure 6 summarizes the role of each agent of *CodeBroker*. In Fig. 6, `T` represents components that can potentially be reused in the current development task. From them, software developers need to choose the most appropriate one. `D1` and `D2` represent delivered reusable components before and after user profiles are considered respectively. Ideally, active reuse repository systems should present software developers the `T` set with black components (known to software developers) removed. However, due to the incompleteness of reuse queries, irrelevant components and missed components are unavoidable. Therefore, retrieval by reformulation is



T is the set of components relevant to the task at hand
D1 is the set of components delivered without consideration of user profiles
D2 is the set of components delivered after user profiles are considered

components that should have been considered by programmers
components that may be located by programmers
components that would not be considered without active delivery
components missed by active delivery
irrelevant components delivered

Figure 6: The role of each agent

needed to allow software developers to move D2 toward T incrementally. Direct manipulation of retrieved results lets software developers remove irrelevant components (circles shaded with waves) quickly; and query refinement let software developers incorporate missed components (circles with no shade) into D2 in following locating efforts with incrementally developed reuse queries.

5. Related Work

5.1. Software Reuse Repository Systems

There has been much research on software reuse repository systems. *CodeBroker* is most similar to those reuse repository systems that adopt the information retrieval approach to the indexing and retrieving of reusable components. Some systems index components based on the associated documents [13]; and some extract information directly from comments and names from source codes [4, 14]. *CodeBroker* is also similar to those systems that use signatures to index reusable components [26]. The greatest advantage of these two retrieval techniques is its low cost in both setting up the repository and posing a query.

Other more complicated retrieval mechanisms have also been proposed. *LaSSIE* [3] and *AIRS* [16] use multi-facets to represent reusable components, and semantic networks to support concept-based retrieval. In *CodeFinder* [10], components are represented with frames and spread activation is used as the retrieval mechanism. Despite the sophistication of those mechanisms, empirical studies have shown their retrieval effectiveness presents no advantage over free-text retrieval systems [9] [15].

Active reuse repository systems like *CodeBroker* are meant to complement those traditional reuse repository systems to assist software developers in reusing unanticipated components. *CodeBroker* distinguishes itself from other systems by providing a new way of interaction with reuse repository systems.

5.2. Information Agents

Active reuse repository systems can be viewed as information agents. The closest work to *CodeBroker* is *Remembrance Agent* that augments human memory by actively presenting documents relevant to the user's current writing [19]. Although its goal is to remind users of those documents that might have been forgotten, it does not differentiate documents that are still remembered from those forgotten. Another similar agent is *Letizia* that assists users in browsing the World Wide Web by suggesting web pages within a few links of the current page [12]. It monitors and analyzes the past browsing activities of a user to build up a profile of user interests. This profile is then used to determine the most related web pages.

In addition to the difference of application domains and information retrieval mechanisms used, *CodeBroker* is also different because it considers both the relevance to the task and the relevance to the user. *Remembrance Agent* seeks for information relevant to the task only and *Letizia* seeks for information relevant to the user only.

6. Summary

Active reuse repository systems reduce the number of interactions and the complexity of locating reusable components. With passive reuse repository systems, developers must consciously decide to start a reuse process, actually initiate the locating process by switching working contexts, and compose and input the appropriate reuse queries. All these actions are automated by active reuse repository systems. Although the components presented by active reuse repository systems may not be accurate enough, it can lead developers into reuse mode without losing touch with their development task, and give them the first round of location results which can be reformulated further.

Although *CodeBroker* is currently 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 deliver potentially reusable components. We started with the coding phase because there are more available reusable components in the code level, and it is relatively easier to evaluate the effectiveness of the concept of active reuse repository systems. As our future research agent, an evaluation of *CodeBroker* will be performed to gain better understanding of the difficulties encountered by software developers when reuse is integrated into their development activities.

7. Acknowledgements

I would like to thank Gerhard Fischer, Brent Reeves and Kumiyo Nakakoji for their valuable feedback on this research. I would also like to thank the anonymous reviewers for their insightful comments.

8. References

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

[2] Buckley, C., G. Salton, and J. Allan, "The Effect of Adding Relevance Information in a Relevance Feedback Environment," In *Proceedings of 17th Annual International ACM SIGIR Conference*, Dublin, Ireland, 1994, pp. 292-300.

[3] Devanbu, P., R. J. Brachman, P. G. Selfridge, and B. W. Ballard, "LaSSIE: A Knowledge-Based Software Information System," *Commun. of the ACM*, 34(5), pp. 34-49, 1991.

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

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

[6] Fischer, G., K. Nakakoji, J. Ostwald, G. Stahl, and T. Sumner, "Embedding Critics in Design Environments," in *Readings in Intelligent User Interfaces*, M. T. Maybury and W. Wahlster, Eds.: Morgan Kaufmann Publisher, 1998, pp. 537-559.

[7] 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, J. Grudin, W. Buxton, and S. Greenberg, Eds., 2nd ed. San Francisco, CA: Morgan Kaufmann Publishers, 1995, pp. 822-831.

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

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

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

[11] 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), pp. 211-240, 1997.

[12] Lieberman, H., "Autonomous Interface Agents," In *Proceedings of Human Factors in Computing Systems 1997*, Atlanta, GA, 1997, pp. 67-74.

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

[14] Michail, A. and D. Notkin, "Assessing Software Libraries by Browsing Similar Classes, Functions and Relationships," In *Proceedings of 21st ICSE*, Los Angeles, CA, 1999, pp. 463-472.

[15] Mili, H., E. Ah-Ki, R. Grodin, and H. Mcheick, "Another Nail to the Coffin of Faceted Controlled-Vocabulary Component Classification and Retrieval," In *Proceedings of Symposium on Software Reuse 1997*, MA, USA, 1997, pp. 89-98.

[16] Ostertag, E., J. Hendler, R. Prieto-Diaz, and C. Braun, "Computing Similarity in a Reuse Library System: An AI-Based Approach.," *ACM Trans. on Software Engineering and Methodology*, 1(3), pp. 205-228, 1992.

[17] Prieto-Diaz, R., "Reuse as a New Paradigm for Software Development," in *Systematic Reuse: Issues in Initiating and Improving a Reuse Program*, M. Sarshar, Ed.: Springer, 1996, pp. 1-13.

[18] Rada, R., *Software Reuse: Principles, Methodologies and Practices*. Norwood, NJ: Ablex Publishing, 1995.

[19] Rhodes, B. J. and T. Starner, "Remembrance Agent: A Continuously Running Automated Information Retrieval System," In *Proceedings of 1st International Conference on The Practical Application of Intelligent Agents and Multi Agent Technology*, London, UK, 1996, pp. 487-495.

[20] Rosenbaum, S. and B. DuCastel, "Managing Software Reuse--An Experience Report," In *Proceedings of 17th ICSE*, Seattle, Washington, 1995, pp. 105-111.

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

[22] Sen, A., "The Role of Opportunism in the Software Design Reuse Process," *IEEE Trans. on Soft. Eng.*, 23(7), pp. 418-436, 1997.

[23] Soloway, E. and K. Ehrlich, "Empirical Studies of Programming Knowledge," *IEEE Trans. on Soft. Eng.*, SE-10(5), pp. 595-609, 1984.

[24] Ye, Y. and G. Fischer, "Promoting Reuse with Active Reuse Repository Systems," In *Proceedings of the 6th International Conference on Software Reuse*, Vienna, Austria, 2000, pp. 302-317.

[25] Ye, Y., G. Fischer, and B. Reeves, "Integrating Active Information Delivery and Reuse Repository Systems," In *Proceedings of 8th ACM International Symposium on Foundations of Software Engineering*, San Diego, CA, 2000, (to appear).

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