Personalizing Delivered Information in a Software Reuse Environment

Gerhard Fischer[1] and Yunwen Ye[1,2]

[1] Center for LifeLong Learning and Design, Department of Computer Science
University of Colorado, Boulder, Colorado 80309-0430, USA
Tel: +1-303-492-1502 Fax: +1-303-492-2844
{gerhard, yunwen}@cs.colorado.edu
[2] Software Research Associates, Inc., 3-12 Yotsuya, Tokyo 160-0004, Japan

Abstract. Browsing- and querying-oriented schemes have long served as the principal techniques for software developers to locate software components from a component repository for reuse. Unfortunately, the problem remains that software developers simply will not actively search for components when they are unaware that they need components or that relevant components even exist. Thus, to assist software developers in making full use of large component repositories, *information access* need to be complemented by *information delivery*. Effective delivery of components calls for the personalization of the components to the task being performed and the knowledge of the user performing it. We have designed, implemented, and evaluated the *CodeBroker* system to support personalized component delivery to increase the usefulness of a Java software reuse environment.

Keywords: Task modeling, discourse modeling, user modeling, software reuse, information delivery.

1 Introduction

Browsing- and querying-oriented schemes have long served as the principal techniques for people to retrieve information in many applications, including systems for locating software components and for exploring the World Wide Web. However, these conventional retrieval techniques do not scale up to large information stores. More innovative schemes, such as *query by reformulation* [18] and *latent semantic analysis* [10], have introduced new possibilities. Unfortunately, the problem remains that *users simply will not actively search for information when they are unaware that they need the information or that relevant information even exists.* Thus, to assist users in making full use of large information repositories, information access methods need to be complemented by information delivery methods.

Evidence exists that the lack of support for information delivery has long been a significant obstacle to the success of software reuse [5, 19]. Before software developers can reuse components, they have to either know the components already or locate them quickly and easily. Component location is often supported by component repository systems, most of which, like other information systems, support browsing and querying only. They fall short in helping software developers who make no attempt to locate components [5]. Even if software developers are aware of the components, they may not be able to retrieve them because of the mismatch between the situation model and the system model [9]. The *situation model* refers to the understanding of the task by developers or users, and the *system model* refers to the names and descriptions of components in repository systems, which are predetermined by system designers. Instead of passively waiting to be discovered, component repository systems need to be more active in supporting reuse by delivering potentially reusable components to software developers when they are engaged in development activities.

2 Information Delivery Systems

High-Functionality Applications. Component repository systems are high-functionality applications (HFAs) [3]; they often include thousands of components and evolve quickly. The design of HFAs must address two problems: (1) the unused functionality must not get in the way, and (2) unknown existing functionality must be accessible or delivered when it is needed.

Most current component repository systems are passive, meaning users (software developers) have to initiate the search process through information access methods—either browsing or querying—to locate components. These passive systems are designed under the assumption that users are aware of their information needs and that they know how to ask for it. Our empirical studies have shown, however, that a user's knowledge on an HFA does not match the system itself. Typically, a user has four levels of knowledge in an HFA:

– level 1 knowledge (L1): elements are well known and regularly used;
– level 2 knowledge (L2): elements are known vaguely and used occasionally;
– level 3 knowledge (L3): elements are believed to exist in the system;
– level 4 knowledge (L4): all elements of the system.

Elements falling in the area L4 – L3 become information islands, and passive systems cannot help users to find them because their existence is not even known [19]. Information delivery, a method by which the system initiates the information search process and volunteers information to users, can build a bridge to the information islands. Moreover, compared with passive systems, information delivery systems make it easier for users to locate information in L3.

The Limitations of Task- and User-Independent Information Delivery Systems. Systems that just throw a piece of decontextualized information at users are of little use because they ignore the working context. The working context consists of the task being performed and the user performing it. The challenge for information delivery is to deliver *context-sensitive*, or *personalized*, information, related to both the task at hand and the background knowledge of the user. Task- and user-independent information delivery systems (or "push" systems) such as Microsoft's "Tip of the Day" suffer from the problem that concepts get thrown at users in a decontextualized way. Despite the possibility for interesting serendipitous encounters of information, most users find this feature more annoying than helpful.

***CodeBroker*—An Active Component Repository System.** We have designed, implemented, and evaluated an active component repository system, named *CodeBroker*, that supports information delivery [20]. *CodeBroker* delivers task-relevant and user-specific components to Java developers by (1) constructing a *task model* to capture the programming task through continuously monitoring the programming activities in a software development environment, (2) identifying the domain of users' current interest by creating a *discourse model* based on the interaction history between the system and the user, and (3) creating a *user model* to represent each user's knowledge about the repository to assure that only unknown components are delivered.

3 Task-Relevant Component Delivery

General Approaches to Task Modeling. Tasks can be modeled through either plan recognition or similarity analysis [4, 6]. The plan recognition approach uses plans to specify the link from a series of primitive user actions to the goal of a task. When actions of a user match the action part of a plan, the system deems the user to be performing the corresponding task, and information about the task is delivered. The similarity analysis approach examines the self-revealing information in the context surrounding the current focus of users and uses that information to predict their needs for new information. The system then delivers information from the repository that has high similarity to the contextual circumstance. Plan recognition-based task modeling systems are difficult to scale up because it is difficult to create plans. Unlike plan recognition, by which the system tries to infer the task goal, the similarity analysis approach matches a task to the information sharing the same contextual circumstances.

Task Modeling in *CodeBroker*. *CodeBroker* adopts similarity analysis to find relevant components. It utilizes the descriptive elements of programs and finds components that have similar descriptions. 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.

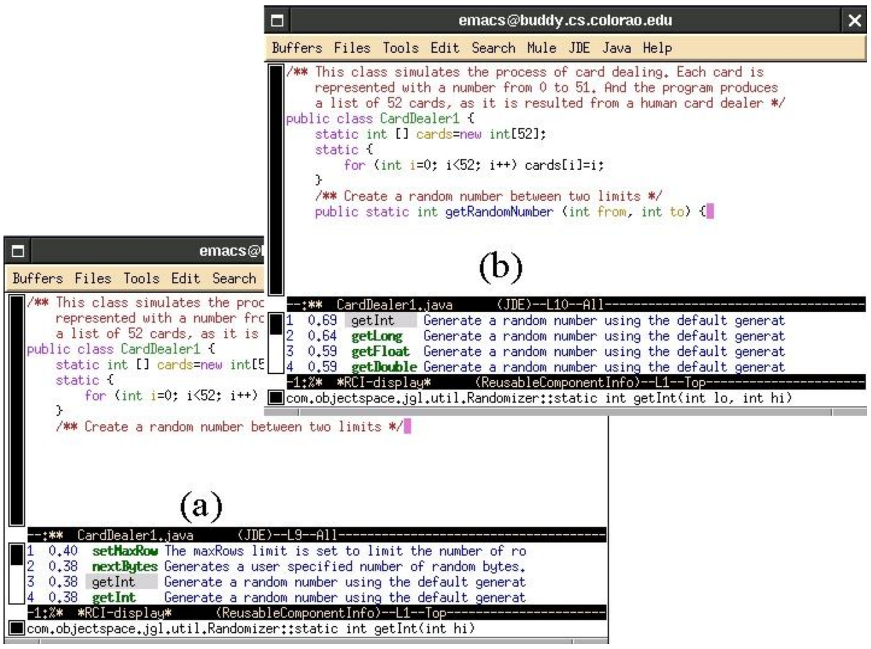
Important concepts of a program are often contained in its informal information, such as comments and identifier names that are important beacons for program comprehension [19]. Modern programming languages such as Java further enforce the inclusion of self-explaining informal information by introducing the concept of *doc comments*. A doc comment begins with "`/**`" and continues until the next "`*/`". Contents inside doc comments describe the functionality of the following module, either a class or a method.

Constraints of a program are captured by its *signature.* A signature defines the syntactic interface of a program by specifying the types of input and output data. For a component to be easily integrated, its signature should be compatible with the environment in which it will be incorporated.

CodeBroker models tasks by extracting doc comments and signatures of the program under development. It tries to find relevant components with similar concepts and constraints. Relevance of components to the task at hand is 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 components in the repository.

Constraint compatibility is the type compatibility existing from the signature of the program under development to the signatures of repository components.

Concept similarity is computed by Latent Semantic Analysis (LSA) [10]. After being trained with a large volume of domain-specific documents, the semantic space created by LSA for that domain can capture the latent semantic association among words, and thus it bridges the gap between the situation model and the system model [9]. Constraint compatibility is computed by signature matching. Two signatures match if their input types and output types are both in structural conformance [19]. Figure 1 shows



emacs@buddy.cs.colorao.edu
Buffers Files Tools Edit Search Mule JDE Java Help
(b)
emacs@
Buffers Files Tools Edit Search
(a)

```
/** 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 */
    public static int getRandomNumber (int from, int to) {
--:**  CardDealer1.java      (JDE)--L10--All-----------------------------------
1  0.69  getInt    Generate a random number using the default generat
2  0.64  getLong   Generate a random number using the default generat
3  0.59  getFloat  Generate a random number using the default generat
4  0.59  getDouble Generate a random number using the default generat
-1:%*  *RCI-display*      (ReusableComponentInfo)--L1--Top--------------------
com.objectspace.jgl.util.Randomizer::static int getInt(int lo, int hi)
```

```
/** This class simulates the proc
    represented with a number frc
    a list of 52 cards, as it is
public class CardDealer1 {
    static int [] cards=new int[5
    static {
        for (int i=0; i<52; 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
-1:%*  *RCI-display*      (ReusableComponentInfo)--L1--Top--------------------
com.objectspace.jgl.util.Randomizer::static int getInt(int hi)
```

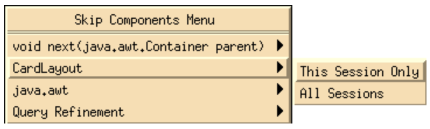
Fig. 1. Task-relevant delivery of components. Components delivered when a doc comment is entered (a) or a signature is defined (b). The third component in (a) has a similar concept but incompatible signature (shown in the mini-buffer). The delivery in (b) is based on the task model including both doc comments and signatures, and its first component matches the task.

an example of task-relevant delivery of components. A programmer wants to create a random number between two integers, and expresses his task in the doc comment: `Create a random number between two limits`. The comment serves as a task model, based on which the system delivers several components (Fig. 1a). However, the task model is not complete because it does not say the method must take two integers as input. When the signature (`int x int -> int`) is defined (Fig. 1b), the system acquires a more precise task model combining both the concept (doc comment) and the constraint (signature), and gives a better delivery. In Fig. 1b, the first delivered component matches the task and can be immediately used.

Developing a Discourse Model through Retrieval by Reformulation. The components delivered solely based on the task model may not be precise enough. First, because the task model is not directly represented—a direct representation is the program code—it is partial and biased. Second, doc comments do not always reflect what software developers want to do. Similarly, descriptions of components in the repository are not complete and precise enough, either.

CodeBroker supports *retrieval by reformulation* [18] to complement the impreciseness of task models. Retrieval by reformulation is a dynamic, interactive information location approach that allows users to develop their queries incrementally [18]. After evaluating the delivered components, software developers can either refine the query or directly manipulate the delivered components.

Through direct manipulation of each delivered component, software developers can tell the system explicitly what does not interest them currently. In *CodeBroker*, each delivered component is associated with the `Skip Components Menu` (Fig. 2), which has three items: the component itself, its class, and its package. If the developer does not want to have the method, or all the components in the class or the package, delivered again in this development session because they are irrelevant to the current project, the developer can choose the appropriate item and then the `This Session Only` command. Then the system will remove the method or all methods from the class or the package from automatic delivery. Direct manipulation can incrementally create, be-

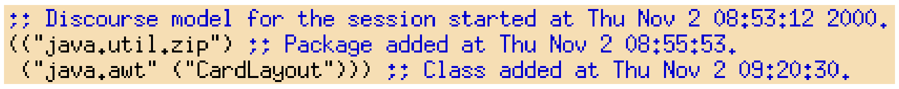


Users can select either the method, its class, or its package, and then select `This Session Only` to add it to the discourse model. The `All Sessions` command adds it to the user model.

Fig. 2. The Skip Components Menu

tween the developer and the system, a shared understanding of the larger context of the current task. Components in a repository are organized in hierarchy according to their application domains (packages) and inheritance relationship. Most projects need only a part of the repository. If the system knows in which part the developer is currently interested, it can deliver more relevant information.

This shared understanding of a developer's current interest is captured in discourse models. A discourse model represents the previous interactions between the user and the system in one development session and serves as a filter to remove irrelevant components in later deliveries in the same development session. It also reduces the delivery of irrelevant components caused by polysemy—a difficult problem for any information retrieval system—by limiting searching domains because polysemous words often have different meanings in totally different domains. For example, if the programming task is to shuffle a deck of cards, the developer may use the word "card" in doc comments. That would make the system deliver components from the class `java.awt.CardLayout`, a GUI class in which "card" means a graphical element. If the project does not involve interface building, this whole class is irrelevant. Developers can add the class or even the whole package (`java.awt`) to the discourse model to prevent components of it from being delivered in the development session.



```
;; Discourse model for the session started at Thu Nov 2 08:53:12 2000.
(("java.util.zip") ;; Package added at Thu Nov 2 08:55:53.
 ("java.awt" ("CardLayout"))) ;; Class added at Thu Nov 2 09:20:30.
```

Fig. 3. A discourse model, which is a Lisp list of items with the format: `(package-name (class-name (method-name)))`. An empty `class-name` or `method-name` area indicates the whole package or class should not be delivered in this development session.

A discourse model in *CodeBroker* is in the format of a Lisp association list (Fig. 3). Every development session starts with an empty discourse model, and its contents are gradually added by developers, during the use of the system, through the `Skip Components Menu` (Fig. 2).

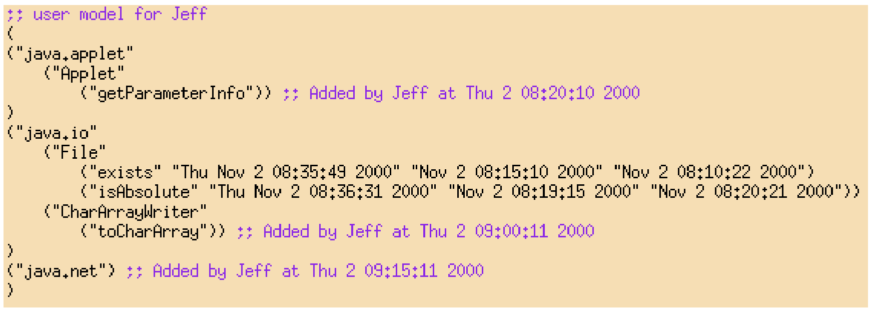
4 User-Specific Component Delivery

Information delivery [3] is meant to inform software developers of those components that they do not know. Therefore, the system needs to know what they know already. We use user models to represent software developers' knowledge about the component repository. User models in *CodeBroker* are both adaptable and adaptive [2, 17].

User Models in *CodeBroker*. A user model in *CodeBroker* contains a list of components known to the software developer (Fig. 4). Each item in the list is a package, a class, or a method. Each component retrieved from the component repository is looked up in the user model before it is delivered. If a method component matches a method in the user model, and the user model indicates that the developer has used it more than three times (adjustable by developers), the system assumes the developer knows it already and removes it from the delivery. If the method has no `use-time`, it means the method was added by the developer, who had claimed he or she had known it very well and did not want it delivered. If the class of the method (which has no method list in the user model), or the package of the method (which has no class list) is included in the user model, the method is also removed.

Adaptable User Models. Software developers can explicitly update their user models through interactions with *CodeBroker*. If they find one known component is delivered, they can click the component to bring up the `Skip Components Menu` (Fig. 2), where they can choose one abstraction level (method, class, or package) and then select the `All Sessions` command. The method (or its class or package) will be added to the user model. User-added components have no `use-time` field (Fig. 4).

Adaptive User Models. Due to the large volume of components and the constantly evolving nature of repository systems, for software developers to maintain their user models is a time-consuming task. Therefore, user models in *CodeBroker* are adaptive. *CodeBroker* continuously monitors developers' actions in the development environment. Whenever a method component from the repository is used by the developer,



```
;; user model for Jeff
(
("java.applet"
    ("Applet"
        ("getParameterInfo")) ;; Added by Jeff at Thu 2 08:20:10 2000
)
("java.io"
    ("File"
        ("exists" "Thu Nov 2 08:35:49 2000" "Nov 2 08:15:10 2000" "Nov 2 08:10:22 2000")
        ("isAbsolute" "Thu Nov 2 08:36:31 2000" "Nov 2 08:19:15 2000" "Nov 2 08:20:21 2000"))
    ("CharArrayWriter"
        ("toCharArray")) ;; Added by Jeff at Thu 2 09:00:11 2000
)
("java.net") ;; Added by Jeff at Thu 2 09:15:11 2000
)
```

Fig. 4. A user model, which is a list of items with the format: `(package-name (class-name (method-name use-time use-time ...)))`. When the use of a component is detected by the system, it is added to the list with the current time as `use-time`. If the component is added by the user, there is no `use-time`. As with discourse models, an empty `class-name` or `method-name` area means the whole package or class is included.

the system adds the component with the current time as `use-time` to the user model. The system adds only methods to the user model; it does not add classes or packages because the use of a class or a package does not mean that the developer knows the whole class or package.

Initialization of User Models. Initial user models are created by analyzing the programs that software developers have written so far. *CodeBroker* analyzes those programs to extract each component used, and if the component is contained in the indexed component repository, it is added to the initial user model with the program modification time as the `use-time`.

5 Related Work

CodeBroker builds upon our previous experience with critiquing systems [4] that model tasks by plan recognition to give feedback to users who have developed a suboptimal solution. *CodeBroker* tries to predict information needs and provide feedforward [16] for users so that they can avoid suboptimal solutions. Some information agents also aim to provide feedforward. For example, Remembrance Agent [15] augments human memory by autonomously displaying old emails and notes relevant to the email being written by the user. Letizia [11] assists users in browsing the WWW by suggesting and displaying relevant web pages. By observing the programmer's Java programming, Expert Finder [11] can refer the programmer to expert helpers who display significant experience in the area in which the programmer is troubled.

Research on component repository systems has mostly focused on the retrieval mechanism only, aiming to improve the relevance of retrieved components to the query submitted by users. The retrieval mechanism of *CodeBroker* is similar to that of those systems using free-text indexing [12]. Many sophisticated retrieval mechanisms have been proposed, such as multifaceted classification, frames and semantic networks, and associative networks [13]. Despite their sophistication and the simplicity of free-text indexing, no significant difference is found in retrieval effectiveness [14]. *CodeBroker* is unique because it is active and it strives to improve the relevance to the task and user, not to the query per se.

6 System Evaluation

To investigate the effectiveness of the system in supporting programmers to reuse by actively delivering personalized components, and to understand the distinctive role that task models, discourse models, and user models play in achieving this personalization, we have conducted 12 experiments with 5 subjects whose expertise in Java programming ranged from medium to expert level. In each experiment, the subject was asked to implement one predetermined programming task with the *CodeBroker* system. Days before the experiments, *CodeBroker* created an initial user model for each subject by analyzing the Java programs that the subject had written recently.

Task models based on similarity analysis in *CodeBroker* were quite effective in delivering task-relevant components. In 10 of the 12 experiments, the subjects reused components delivered by the system. The 12 programs created reused 57 distinct components (if a component was used more than once in a program, it was counted as one), and 20 of them were delivered by the system. Of the 20 reused components delivered by the system, the subjects had not known the existence of nine components. In other words, those nine components were from information islands (see Sect. 2), and they could not have been reused without the support of *CodeBroker*. Although subjects somehow anticipated the existence of the other 11 components (belonging to L3, as discussed in Sect. 2), they had known neither the names nor the functionality, and they had never used them before. They might have reused those 11 components if they could manage to locate those components by themselves. In follow-up interviews, all subjects acknowledged that *CodeBroker* made locating those anticipated components much easier and faster.

Discourse models (Fig. 3) improved the task-relevance when they were created by subjects. In five experiments, subjects used the `Skip Components Menu` (Fig. 2) to create discourse models, which removed a total of 10% of retrieved components from the deliveries. All the components removed by discourse models were not relevant to the programming tasks. The programming tasks of these experiments were rather small, and each development session was short. We expect discourse models to improve the task-relevance of delivered components even further in real programming tasks, which require more interactions between programmers and the system.

User models (Fig. 4), however, removed only 2% of the retrieved components. The following two reasons might have belittled the role of user models in the experiments: (1) initial user models of subjects were not complete because many of the programs that the subjects had written for companies were not available for the creation of initial user models; (2) to observe the effectiveness of task models, subjects were assigned tasks that involved the part of the repository that they did not know well, and as a result, most retrieved components were not known to them and were not included in their user models. Nonetheless, in follow-up interviews, subjects said that they did not notice the delivery of too many components they already knew. A careful examination of those components removed by user models showed that those removed components could not be reused in the tasks. User models helped and are needed to reduce the number of irrelevant components to be delivered, although more experimental data based on long-term use [8] of the system are needed to investigate the contributions of user models.

Overall, the subjects rated the system an average of 6.9 on a scale from 1 (totally useless) to 10 (extremely useful).

7 Summary and Future Research

This paper presents a new approach—component delivery—to assist software developers in reusing components from large repositories. The approach combines task models, discourse models, and user models to improve the context relevance of delivered components. Task-relevant components are first retrieved based on task models, and the retrieval results are the same for all developers and development sessions. The task relevance is further improved by discourse models to reflect the difference of the larger context. Moreover, user models make the delivery specific to users so that different users may be given different information, depending on their particular experiences. Evaluations of the *CodeBroker* system, developed based on this approach, have illustrated each model's contribution to the context relevance.

The challenge in an information-rich world (where human attention is the most valuable and scarcest commodity) is not only to make information available to people at any time, at any place, and in any form, but to reduce information overload by making information relevant to the task-at-hand and to the background knowledge of the users [3]. Information delivery methods will be successful and adopted by users only if they are able to achieve the right balance between the costs of intrusive interruptions and the loss of context-sensitivity of deferred alerts [7].

Currently, the repository is located in the same machine as the development environment of developers and is created statically before its use because most current repositories are closed and proprietary. As the movement of Open Source Systems attracts more and more developers, we can expect more software components to become open-source, for example, the Jun system (a 3D Smalltalk/Java library) [1]. It will then become increasingly difficult for software developers to know newly available open-source components. We envision a distributed *CodeBroker* system running on several computers where software developers contribute open-source components. The system will dynamically index components from those constantly evolving repositories and then will deliver components through networks to other developers. Through the mediation of *CodeBroker*, software developers can thus benefit from each other's work and improve the productivity of software development through avoiding unnecessary repetition of work.

Acknowledgments. The authors thank the members of the Center for LifeLong Learning & Design at the University of Colorado, who have made major contributions to the conceptual frameworks described in this paper. Kumiyo Nakakoji provided many helpful comments. The research was supported by (1) the National Science Foundation, Grants REC-9631396 and IRI-9711951; (2) Software Research Associates, Tokyo, Japan; (3) PFU, Tokyo, Japan; and (4) the Coleman Family Foundation, San Jose, CA.

References

1. A. Aoki, K. Hayashi, K. Kishida, K. Nakakoji, Y. Nishinaka, B. Reeves, A. Takashima, and Y. Yamamoto. A case study of the evolution of Jun: An object-oriented open-source 3D multimedia library. In *Proc. of 23rd International Conference on Software Engineering* (to appear), Toronto, Canada, 2001.
2. G. Fischer. Shared knowledge in cooperative problem-solving systems—integrating adaptive and adaptable components. In M. Schneider-Hufschmidt, T. Kuehme, and U. Malinowski, (eds.), *Adaptive User Interfaces: Principles and Practice*, pp. 49–68. Elsevier Science, Amsterdam, 1993.
3. G. Fischer. User modeling in human-computer interaction. *User Modeling and User-Adapted Interaction* (to appear), 2001.
4. G. Fischer, K. Nakakoji, J. Ostwald, G. Stahl, and T. Sumner. Embedding critics in design environments. In M. Maybury and W. Wahlster, (eds.), *Readings in Intelligent User Interfaces*, pp. 537–559. Morgan Kaufmann, 1998.
5. W. Frakes and C. Fox. Quality improvement using a software reuse failure modes model. *IEEE Transactions on Software Engineering*, 22(4):274–279, 1996.
6. B. Gutkauf. *Improving Design & Communication of Business Graphs through User Adaptive Critiquing*. Ph.D. Dissertation, Universitat-GH Paderborn, Paderborn, Germany, 1998.
7. E. Horvitz, A. Jacobs, and D. Hovel. Attention-sensitive alerting. In *Proc. of Conference on Uncertainty and Artificial Intelligence 1999*, pp. 305–313, San Francisco, CA, 1999.
8. J. Kay and R. Thomas. Studying long-term system use. *CACM*, 38(7):61–68, 1995.
9. W. Kintsch. *Comprehension: A Paradigm for Cognition*. Cambridge University Press, 1998.
10. T. Landauer and S. 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.
11. H. Lieberman. Personal assistants for the web: An MIT perspective. In M. Klusch, (ed.), *Intelligent Information Agents: Agent-Based Information Discovery and Management on the Internet*, pp. 279–292. Springer-Verlag, 1999.
12. Y. Maarek, D. Berry, and G. Kaiser. An information retrieval approach for automatically constructing software libraries. *IEEE Transactions on Software Engineering*, 17(8):800–813, 1991.
13. A. Mili, R. Mili, and R. T. Mittermeir. A survey of software reuse libraries. In W. Frakes, (ed.), *Systematic Software Reuse*, Annals of Software Engineering 5, pp. 317–347. Baltzer Science, Bussum, The Netherlands, 1998.
14. H. Mili, E. Ah-Ki, R. Grodin, and H. Mcheick. Another nail to the coffin of faceted controlled-vocabulary component classification and retrieval. In *Proc. of Symposium on Software Reuse*, pp. 89–98, Boston, MA, 1997.
15. B. Rhodes and T. Starner. Remembrance agent: A continuously running automated information retrieval system. In *Proc. of 1st International Conference on the Practical Application of Intelligent Agents and Multi Agent Technology*, pp. 487–495, London, 1996.
16. H. Simon. *The Sciences of the Artificial*. The MIT Press, 3rd edition, 1996.
17. C. Thomas. *To Assist the User: On the Embedding of Adaptive and Agent-Based Mechanisms*. Oldenbourg Verlag, Munich, 1996.
18. M. Williams. What makes RABBIT run? *International Journal of Man-Machine Studies*, 21:333–352, 1984.
19. Y. Ye and G. Fischer. Promoting reuse with active reuse repository systems. In *Proc. of 6th International Conference on Software Reuse* , pp. 302–317, Vienna, Austria, 2000.
20. Y. Ye. *Supporting Component-Based Software Development with Active Component Repository Systems*. Ph.D. Dissertation, University of Colorado, Boulder, CO, 2001.