2002 International Conference on Intelligent User Interface (IUI'02), San Francisco, CA, (to appear), Jan. 13-16, 2002

Information Delivery in Support of Learning Reusable Software Components on Demand

Yunwen Ye[1,2]
[1]SRA Key Technology Laboratory, Inc.
3-12 Yotsuya, Shinjuku
Tokyo 160-0004, Japan
yunwen@cs.colorado.edu

Gerhard Fischer[2]
[2]Department of Computer Science
University of Colorado
Boulder, CO 80309-0430, USA
gerhard@cs.colorado.edu

Abstract

An inherent dilemma exists in the design of high-functionality applications (such as repositories of reusable software components). In order to be useful, high-functionality applications have to provide a large number of features, creating huge learning problems for users. We address this dilemma by developing intelligent interfaces that support learning on demand by enabling users to learn new features when they are needed during work. We support learning on demand with information delivery by identifying learning opportunities of which users might not be aware. The challenging issues in implementing information delivery are discussed and techniques to address them are illustrated with the CodeBroker system. CodeBroker supports Java programmers in learning reusable software components in the context of their normal development environments and practice by proactively delivering task-relevant and personalized information. Evaluations of the system have shown its effectiveness in supporting learning on demand.

Keywords

Information delivery, learning on demand, software reuse, user models, Java programming.

INTRODUCTION

Today's computing world is full of *high-functionality applications (HFAs)* that contain thousands of features and large volumes of useful information [8]. Mastering such HFAs presents huge learning challenges for users. Users cannot learn all the features of an application before they start using it. Also, users, who use an application as a tool, do not have to know all the features because few of them want to become an expert of the application per se. A practical approach is for users to *learn on demand* [7]. Users learn to use a new feature or acquire new information when it is needed during work. Learning on demand is a promising approach because (1) it contextualizes learning by integrating it into work rather than relegates it to a separate phase, and (2) it lets users see for themselves the usefulness of new information for authentic problem situations, thereby increasing the motivation for learning.

Computer systems with interfaces that support learning on demand have to address several challenging issues [8]:

- Users may not be aware of the existence of new features or information.
- Users may not be motivated to learn if they think learning requires too much time and effort.
- Users may not be able to find the new feature or information.
- User may not be able to understand and apply the new feature or information.

Information delivery mechanisms that proactively provide users with task-relevant and personalized information during the work situation present promising solutions to the above issues. We discuss the necessities and challenges in supporting learning on demand with information delivery, and use Java programming as an example domain to illustrate how to design interfaces that assist programmers in learning reusable software components (classes or methods) in the context of their normal programming environments and practice.

A component repository system that supports software reuse by helping programmers locate, comprehend, and modify components [10] has three connotations: a repository that contains components, an indexing and retrieval mechanism, and an interface for user interaction. In viewing component repository systems as examples of HFAs, we are primarily concerned with designing intelligent interfaces for component repository systems that can seamlessly integrate the learning and reuse of components with the normal programming environments and practice. We have been developing a system called *CodeBroker* [26], which motivates and helps programmers to learn and reuse components. By observing programmers' activities in a programming environment, *CodeBroker* infers the needs for components [11, 19] and proactively delivers components that match the inferred needs along with examples that show how the components can be reused to support the programmer in learning about the delivered components.

INFORMATION USE IN HFAS

This section discusses the necessities and benefits of information delivery in supporting learning on demand in HFAs.

Four Levels of Knowledge

Our empirical studies have shown that users typically have different levels of knowledge about HFAs [8]. In Figure 1, rectangle L4 represents the actual information repository, and the ovals (L1, L2, and L3) represent a particular user's different levels of knowledge of the information repository of an HFA. L1 represents the elements that are well known and can be easily used without consulting help or documentation systems. L2 contains the elements that users know vaguely. L3 contains the elements that users anticipate to exist.

Four Modes of Information Use

Depending on their knowledge, users can use a piece of information in an HFA in four different modes: *use-by-memory*, *use-by-recall*, *use-by-anticipation*, and *use-by-delivery*. Different information acquisition approach is needed in each mode.

Use-by-Memory. In the use-by-memory mode, users directly apply the information they have learned before in their work (i.e., information in L1 of Figure 1). No particular tool support is needed in this mode because users already have all the needed knowledge.

Use-by-Recall. In the use-by-recall mode, users vaguely recall that some useful information exists in the system (i.e., information in L2), but they do not remember exactly what it is and how to use it. They need to search the system or the documentation of the system to find what they need. In this mode, users are determined to find the needed information because they know they can benefit from it.

Use-by-Anticipation. In the use-by-anticipation mode, users anticipate that some useful information might be available in the system (i.e., information contained in L3). In this mode, if users cannot find what they want quickly enough, they may give up the learning opportunity and try to solve their problems without taking advantage of the information repository.

Use-by-Delivery. In the previous three modes, users initiate the information-seeking process. However, users will not seek for the information that falls in area (L4 – L3), whose existence is not anticipated. Information delivery is needed to help users acquire information contained in (L4 – L3).

Information Delivery

Information delivery supports not only the use-by-delivery mode but also the use-by-anticipation and use-by-recall modes. In the use-by-recall and use-by-anticipation modes, users are aware of the learning opportunity, and therefore they might initiate the learning process themselves. However, users might miss this learning opportunity if they perceive that locating the relevant information costs too much effort and time to justify its potential value, or are unable to find the relevant information with browsing and querying.

A *productivity paradox* [3] exists in the use of HFAs. Even though more effective strategies of solving a problem exist, most users are not motivated to learn the new strategies. They will "play it safe" by creating a suboptimal solution with their more "primitive" skills because they do not want to spend the extra time and effort required to locate the relevant information.

Even if users are willing to locate the relevant information, they might not be able to do so with browsing and querying mechanisms. Browsing requires that users have a fairly good understanding about the structure of the information repository, and it is not scalable. Querying requires that users be able to formulate a well-defined query that clearly states their information needs, which is a cognitively challenging task [10].

Information delivery, which requires neither user-initiated information-seeking processes nor user-provided queries,

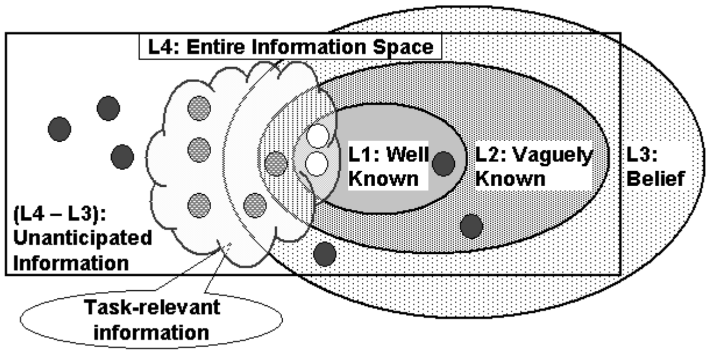

L4: Entire Information Space
L1: Well Known
L2: Vaguely Known
L3: Belief
(L4 – L3): Unanticipated Information
Task-relevant information

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

The essential challenge in supporting learning on demand with information delivery is how to identify the task-relevant information a user does not know (shaded dots). The cloud represents the information needed for the *inferred* task-at-hand (with fuzzy boundaries because the system has only a partial understanding of it). The black dots are not relevant and should therefore not be delivered. The white dots inside the cloud should not be delivered because they are already known by the user.

enables users to skim or perceive task-relevant information at a glance instead of going through a lengthy browsing or querying process. The mitigation of effort and time required for information seeking makes learning on demand a less motivationally demanding task [3].

CODEBROKER: SUPPORTING LEARNING COMPONENTS ON DEMAND

We are investigating the issues about designing intelligent interfaces that support learning on demand in the domain of object-oriented programming. Given the large number and the ever-changing nature of reusable components in object-oriented programming languages, learning on demand is the only viable way for programmers to learn components. The problem is when programmers realize they need to learn a new component and how they can locate and comprehend it. We are developing a system called *CodeBroker* (Figure 2) that can help programmers identify such component learning opportunities with the support of an information delivery mechanism.

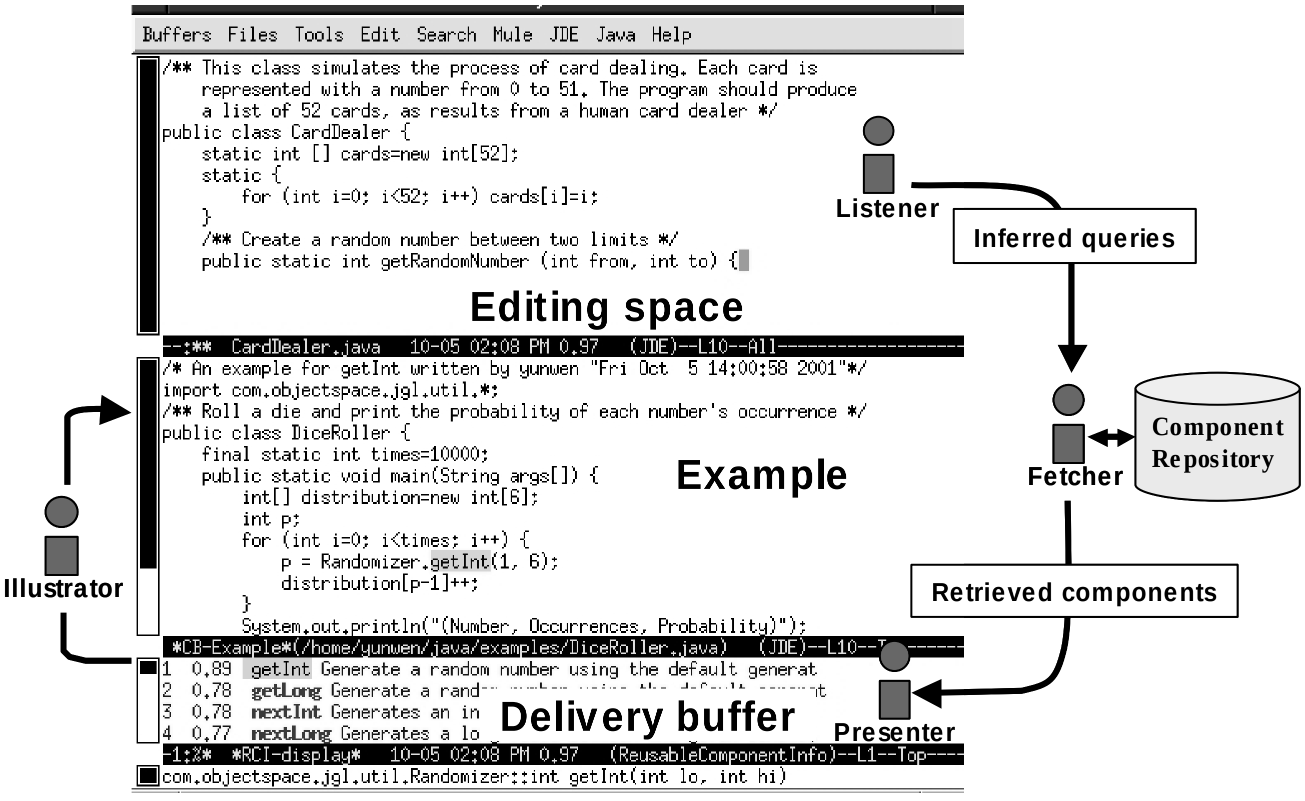
Overview of CodeBroker

The *CodeBroker* system assists programmers to learn components on demand in two ways. First, it proactively delivers components that programmers do not yet know but that are potentially reusable in the current programming task by analyzing the partial programs under development in a programming environment (*Emacs*). Second, it locates example programs that use the component programmers want to learn.

CodeBroker consists of four software agents: *Listener*, *Fetcher*, *Presenter*, and *Illustrator*. *Listener* infers the needs for new components from the partially written program in *Emacs* and creates a reuse query based on the inferred needs. The reuse query is passed to *Fetcher*, which searches the repository to return a set of components that match the query. *Presenter* delivers the matching components to the programmer after it has removed components that are already known to the programmer. Programmers who want to see an example program that uses the component of interest can invoke *Illustrator*.

Listener

Running continuously in the background of *Emacs*, *Listener* predicts the needs for components by analyzing the information contained in partially written programs. The prediction is made by the *similarity analysis* approach [26], which assumes: "If the current working situation, defined by the information in the workspace, is similar enough to a previous situation in which information X was used, then it



```
/** This class simulates the process of card dealing. Each card is
    represented with a number from 0 to 51. The program should produce
    a list of 52 cards, as results from a human card dealer */
public class CardDealer {
    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) {
--:**  CardDealer.java    10-05 02:08 PM 0.97    (JDE)--L10--All------------------
/* An example for getInt written by yunwen "Fri Oct  5 14:00:58 2001"*/
import com.objectspace.jgl.util.*;
/** Roll a die and print the probability of each number's occurrence */
public class DiceRoller {
    final static int times=10000;
    public static void main(String args[]) {
        int[] distribution=new int[6];
        int p;
        for (int i=0; i<times; i++) {
            p = Randomizer.getInt(1, 6);
            distribution[p-1]++;
        }
        System.out.println("(Number, Occurrences, Probability)");
 *CB-Example*(/home/yunwen/java/examples/DiceRoller.java)   (JDE)--L10--T
1  0.89  getInt Generate a random number using the default generat
2  0.78  getLong Generate a rand
3  0.78  nextInt Generates an in
4  0.77  nextLong Generates a lo
-1:%*  *RCI-display*   10-05 02:08 PM 0.97   (ReusableComponentInfo)--L1--Top----
com.objectspace.jgl.util.Randomizer::int getInt(int lo, int hi)
```

Figure 2: A screen image of *CodeBroker* and its architecture

The top buffer is the editing space, the middle buffer is an example program and is shown only when the programmer activates it, and the bottom buffer displays delivered components. In this example, the developer wants to write a method that creates a random number between two integers, and describes the task in the doc comment and signature before the cursor, based on which several components are delivered in the delivery buffer. The first of those delivered components, `getInt`, is a perfect match and can be reused immediately. An example program that uses `getInt` is shown in the middle buffer.

is highly possible that information X is also needed in the current situation."

To define two programming situations as similar, *CodeBroker* examines the three aspects of a program: concept, code, and constraint. The *concept* of a program is its functional purpose, the *code* is the embodiment of the concept, and the *constraint* is the environment in which it runs. Important concepts of a program are often contained in its informal information structure, such as comments and identifier names, which are important beacons to understanding programs. The concept of *doc comments* in Java is specifically introduced to improve the comprehensibility of programs. Doc comments describe the functionality of the following methods or classes and are used as their documents. One important constraint of a program is its type compatibility, which is manifested in its *signature* [27]. For a component to be easily integrated, its signature should be compatible with the environment into which it is going to be incorporated.

Based on the assumption of *similarity analysis*, a component is highly likely to be reused if it shows either *conceptual similarity* (similarity exists between the doc comment of the program under development and a repository component's textual document, which is also created from its doc comment by *Javadoc*, a utility of the Java Development Kit), or *constraint compatibility* (compatibility exists between the signature of a component and that of the program under development), or both.

Whenever a doc comment is entered in the editor, *Listener* extracts its contents to create a concept query based on which a set of matching components are delivered. If programmers want to give *CodeBroker* more information about their task so they can get a set of more task-relevant components, they can continue to declare the signature of the method. Upon the completion of the signature declaration in the editor, *Listener* extracts its contents to create a constraint query. *CodeBroker* then delivers a set of components that match both the previously extracted concept query and the newly extracted constraint query.

Fetcher

Fetcher performs the retrieval process. It uses a combination of *Latent Semantic Analysis (LSA)* [14] and s*ignature matching* [27] as its retrieval mechanism.

LSA determines the conceptual similarity between a concept query (extracted from doc comments) and the documents of repository components. LSA is a free-text indexing and retrieval technique that takes semantics into consideration. *Signature matching* determines the constraint compatibility between a constraint query (extracted from signatures) and the signatures of repository components. Details about these mechanisms can be found in [26].

Presenter

Presenter decides when and how to present to programmers the components retrieved by *Fetcher*. Components are presented, according to their similarity value computed by *Fetcher*, in the delivery buffer (Figure 2) immediately after the programmer has entered a doc comment or a signature. Programmers can customize the system to determine how many components are delivered. Each delivered component is accompanied by its rank, its similarity value, its name and its synopsis. To avoid interrupting programmers who are not interested in the delivered components, no response to the delivery is required, so programmers can just ignore it.

Presenter does not deliver all components retrieved by *Fetcher*; it uses *discourse models* to remove task-irrelevant components and *user models* to remove user-known components from retrieved components [12].

Java components are organized in packages and classes according to their application domains, and most programming tasks do not involve all the classes and packages. If the system knows the classes and packages that are not involved, it can limit its search to those of interest and improve the task-relevance of delivered components. *Discourse models* provide a way for programmers to specify which classes and packages they are *not* interested in learning in the current programming session. During their interactions with *CodeBroker*, if programmers find a delivered component is from a package or a class of no interest, they can add that class or package to the *discourse model*, and all components from that class or package will not be delivered again by *CodeBroker.*

User models contain components that individual programmers know (i.e., L1 in Figure 1) and therefore should not be delivered for learning. *CodeBroker* creates initial user models for programmers by analyzing the Java programs they have written and extracting all the repository components used repeatedly (e.g., more than three times) in the programs. During their interactions with *CodeBroker*, programmers can explicitly update their user models by adding new components to them when they find known components are delivered. User models are also implicitly updated by *CodeBroker*, which adds a component to the user models when it detects that programmers reuse the component in the editor.

Illustrator

The information about a component delivered by *Presenter* in the delivery buffer is concise and is intended for programmers to decide whether a component is relevant to their tasks and therefore needs to be learned. Programmers who want to learn a component can use *Illustrator* to find more detailed information, such as its complete document, an example use of it, and its source code.

Left-clicking a delivered component launches an external HTML browser that shows the whole document for the component. The whole document is extracted from doc comments in Java source programs by *Javadoc*.

Programmers who want an example use of the component can place the cursor on the delivered component, and activate an added *Emacs* command that autonomously searches a specified list of directories to find a Java program that uses the component. The directories to be searched include the directories in which colleagues of the programmer store the programs that they agree to share.

The example program is presented in the editor (Figure 2), with an added first line that shows who wrote the program, so programmers who need further assistance know to whom they should turn for help. Programmers who are willing to contribute a little bit can activate another added *Emacs* command to rate the example based on its usefulness in illustrating the use of the component. The rating, which is interpreted as peer recommendation [1], goes to a database and is used by *Illustrator* to determine which example should be presented when several examples that use the same component exist. If ratings are available, *Illustrator* chooses the example with the highest average rating; otherwise, it chooses the most simple example program. In its current implementation, *Illustrator* equates the simplicity to the byte-length of Java programs, but it can be extended to use complexity metrics for object-oriented programs [4].

Evolving Component Repository by Programmers

Many component repositories, especially those created in-house, come with source code. Programmers may need to modify the source codes of components if they cannot find one that completely fits their current needs, or if they find some bugs in the components. We are extending the *CodeBroker* system to enable programmers to access the source code of a component with one command, as well as to explore new mechanisms and supporting interfaces that encourage and enable programmers to contribute the modified components back into the repository so that other programmers can reuse them [21]. The two essential challenges we face in supporting this continuous evolution [9] of component repositories by programmers are: (1) how to minimize the extra effort required of contributing programmers by automating the contribution process as much as possible, and (2) how to assure the quality of modified components.

SUPPORTING LEARNING ON DEMAND WITH INFORMATION DELIVERY

This section describes a conceptual framework, derived from our experience with the *CodeBroker* system, for designing information delivery systems in support of learning on demand. The major challenges in implementing such systems are: how to identify the task-relevant information that is not yet known to the user (the shaded dots in Figure 1) [12], and when and how to deliver the information to strike a balance between its value versus its interruption of the user's workflow [13].

Task Relevance

Information delivery systems that just present a piece of de-contextualized information, such as Microsoft's "Tip of the Day," are of little use. Despite the possibility for serendipitous learning opportunities, most users find the decontextualized information more annoying than helpful [8]. To deliver contextualized information that is relevant to the task-at-hand, systems must infer the needs for information from low-level user activities [19]. Plan recognization [2] and similarity analysis [26] are two approaches to doing so. 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 assumes the user is performing the corresponding task, and information about the task is delivered. The *similarity analysis* approach, which is adopted in *CodeBroker*, exploits the information contained 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 that has high similarity to the contextual circumstance.

Personalization

Because different users have differing knowledge about a system, and they do not need to learn what they have known already, information delivery should not return the same set of information to all users. To personalize the located information to the specific background of each user *CodeBroker* employs *user models* [12] to represent the existing knowledge that individual users have of the system.

Delivery Timing

Depending on the temporal order between delivered information and its use, information delivery systems can provide *feedforward* or *feedback* to users.

Each user action has a period of time called *action-present* [25], in which users have decided what to do but have not yet executed the needed operations to change the situation. Information delivered in this period of time is *feedforward* information because it can make users change the course of action after they have learned it. Users who learn from feedforward can avoid constructing a suboptimal solution. *CodeBroker* delivers components as feedforward so programmers can immediately take advantage of a delivered component after they have learned it.

Feedback information is delivered when the action has been finished. Feedback can create a situational backtalk of the action by pointing out a potential breakdown the user did not know, or augment the situational backtalk to help users reflect better on the action just completed [18]. Users who learn from feedback can improve their later performance or modify the problematic action if it can be undone.

Intrusiveness

Because delivered information is unsolicited, it risks interrupting the workflow of users whose primary goal is the task-at-hand rather than learning. Indiscreetly delivered information becomes intrusive and disrupts users' workflow. Information delivery systems need to achieve the right balance between the cost of intrusive interruptions and the loss of context-sensitivity of deferred alerts [13] by carefully considering when and how to deliver information so that it can be best utilized by users. *CodeBroker* delivers components in a low intrusive way because it requires no user response and its delivery window is placed outside of the focal window of programmers [26].

Explanations

Most documents for computer systems describe abstractly what the functionality is, but not how to use it; therefore, users often have difficulty in applying the delivered information. Examples of its use contextualize the abstract concepts of documents and explain to users the expected effect in an intuitive way. They provide useful aids for users to learn how to use, adapt, and combine the new information in their current task by drawing an analogy between the current task and the examples [11]. To fully support the learning of new components, *CodeBroker* locates examples from the programs created and recommended by peer programmers.

Evolving Information Repository

The information needed to support learning on demand cannot be fully covered at the design time of systems. Because system designers are unable to anticipate and provide explanations for all the possible use scenarios of systems, systems must be designed as open systems that evolve at the hands of users [9]. When users experience breakdowns in using the systems and insufficient support from the information repository of the systems, they should be able to report, react, and resolve those problems. Systems, at the same time, should be able to capture and accumulate the emergent expertise from users and share it with other users. In *CodeBroker*, the collection of example programs is evolved by programmers with little extra effort (rating examples), and we are extending *CodeBroker* to support the evolution of component repositories at the hands of programmers.

SYSTEM ASSESSMENT

We have conducted 12 experiments with five programmers to investigate whether programmers are able to learn and reuse components on demand with the support of information delivery. The component repository used in the experiments contained 673 classes and 7,338 methods from several standard Java API libraries.

In each experiment, the subject was asked to implement a task with *CodeBroker*. Based on the subject's current knowledge about the component repository, which we obtained by analyzing the subject's recently written Java programs, we assigned a task that could be implemented either easily with some components in the repository that the subject had not yet known, or less straightforwardly with the subject's current knowledge.

Findings of Experiments

Table 1 summarizes the overall results of the experiments. Subjects learned and reused components delivered by *CodeBroker* in 10 of the 12 experiments. The 12 programs created by subjects used 57 distinct components, 20 of which were delivered by *CodeBroker*.

Of the 20 reused components that were delivered, the subjects did not anticipate the existence of 9 (see column 5 in Table 1). In other words, those 9 components could not have been learned and reused without the support of *CodeBroker*, and the subjects would have created their own solutions instead. Although the subjects anticipated the existence of the other 11 components (see columns 6 and 7 in Table 1), they had known neither the names nor the functionality, and had never reused them before. They might have learned to reuse the 11 components if they could manage to locate them by themselves. In interviews, subjects acknowledged that *CodeBroker* made locating the components much easier and faster, and the reduced difficulty motivated them, as one subject put it, "to take more time to see whether [a component] existed or not."

CodeBroker not only assisted subjects in learning and reusing components right off the deliveries, but also created a *snowball effect* that triggered them to learn other unknown components that were not directly delivered but were needed to reuse those delivered components (see the last column in Table 1). To reuse one component often requires the reuse of other supplementary components that are coupled through parameter passing or accessing the common class variables. In the experiments, when those supplementary components were not known, subjects used the deliveries of *CodeBroker* as the starting point and followed the hyperlinks of the Java documentation system to learn and reuse them.

Regarding the intrusiveness of the deliveries, subjects said, when interviewed, that they were not distracted by the

Table 1: Overall results of experiments

Subject	Experiment no.	Total no. of distinct components reused	No. of distinct components reused from deliveries	Breakdown of reused components from deliveries			No. of reused components triggered by deliveries
				Unanticipated (L4-L3)	Anticipated but unknown (L3)	Vaguely known (L2)	
S1	1	10	4	2	2	0	0
	2	3	1	1	0	0	1
S2	3	7	1	1	0	0	0
	4	4	1	1	0	0	0
	5	5	3	0	2	1	1
S3	6	5	2	1	1	0	1
	7	4	3	1	2	0	1
	8	3	0	0	0	0	0
S4	9	4	3	0	3	0	0
	10	3	1	1	0	0	2
S5	11	4	1	1	0	0	2
	12	5	0	0	0	0	0
Sum		**57**	**20**	**9**	**10**	**1**	**8**

deliveries. They turned their attention to the delivery buffer only when they wanted to find something useful there. Although *CodeBroker* delivers components both at the entering of a doc comment and the declaration of a signature, we noticed that most subjects paid attention only to the deliveries based on doc comments and were not aware of the changes of the delivered components activated by the declaration of signatures. This observation raised two interesting points: (1) a carefully designed interface can reduce the intrusiveness of information delivery to a minimum; and (2) the completion of entering a doc comment is probably the boundary of the *action-present* [25] period for programming, during which time programmers are still planning their programming actions and are willing to explore alternative solutions by learning new components, and after which time they are less inclined to learn because they have already shifted into the stage of action and committed to one chosen solution.

System Extension Based on Evaluations

At the time of the experiments, the functionality of locating example programs (the *Illustrator* agent) was not yet supported. Some components in the repository included simple examples to explain their usage. During the experiments, we noticed that whenever subjects found such examples, they immediately jumped to read them instead of the descriptive texts. However, only very few components are accompanied by examples, and creating examples for each component is a time-consuming task that increases the difficulty and cost of setting up a component repository. As a response to this issue, we added the *Illustrator* agent to explore the decentralized approach [9, 21] to enrich the component repository by supporting the location of examples developed by peer programmers.

RELATED WORK

Most reusable component repository systems [10, 17] have been developed as standalone systems. *CodeBroker* distinguishes itself by using the information delivery mechanism to create a seamless integration between learning components and programming.

A number of intelligent information systems support information delivery. *Remembrance Agent* [22] delivers old emails and notes relevant to the email being written by the user. *Letizia* [16] assists users in browsing the WWW by suggesting and displaying relevant web pages. *XLibris* [5] automatically retrieves, aggregates, and presents information from the Internet about books borrowed by users from a library. *Stamping Advisor* [15] shares our view in stressing the importance of creating seamless interaction between acquiring new information and applying the information in the normal work process by automatically providing previous design examples for reuse.

CodeBroker is similar to the example-based programming environment proposed by Neal [20], which provides examples that can be directly reused or illustrate the syntactical constructs of programming languages and the implementations of algorithms, and the cliché-based programming environment *KBEmacs* [23], which has a repository of program clichés that programmers can reuse. However, in both systems, programmers have to locate examples or clichés by themselves.

Several recent research efforts on intelligent programming environments incorporate information delivery mechanisms. Drummond et. al [6] add to a component browsing system an agent that infers the search goal of programmers by observing their browsing actions and delivers components that match the inferred goal. The *Argo* design environment [24] is equipped with computer critics [11] that deliver general software design knowledge for programmers to reflect upon their current design.

CONCLUSIONS

Complete coverage of the needed knowledge in using HFAs is impossible. Designers of HFAs should not only focus on developing numerous features but also provide assistance for users to learn needed features on demand during their work. Supporting learning on demand with information delivery helps users identify learning opportunities and supports their learning process. The major contribution of this paper is that it identifies the challenging issues in creating intelligent interfaces that support learning on demand with information delivery, and illustrates how to address such issues in the *CodeBroker* system (Table 2). Although many of the concrete solutions adopted in *CodeBroker*, such as signature matching and the contents of user models, depend on the particular domain, the general conceptual framework we have developed are applicable to other domains as well.

Table 2: Issues in supporting learning on demand with information delivery and their solutions in *CodeBroker*

Challenging Issues	**Solutions in *CodeBroker***
Task-relevance	Similarity analysis with LSA and signature matching
Personalization	User models
Delivery timing	Feedforward at action-present
Intrusiveness	Noninterruptive, low-profile delivery
Explanations	Automatically located examples
Evolving information repository	Leveraging peer programmers' programs

ACKNOWLEDGMENTS

We thank Bradley Rhodes, who released the *Remembrance Agent* system under GPL, which was partially reused in *CodeBroker*. We thank the members of the Center for LifeLong Learning & Design at University of Colorado, who have made major contributions to the conceptual frameworks described in this paper. The research was supported by National Science Foundation, Grant REC-0106976; SRA Key Technology Laboratory, Tokyo, Japan; and Coleman Family Foundation, San Jose, CA.

REFERENCES

1. Balabanovic, M., and Shoham, Y. Fab: Content-Based, Collaborative Recommendation. *Commun. ACM*, 1997. **40**(3):66-72.
2. Carberry, S. Techniques for Plan Recognition. *User Modeling and User-Adapted Interaction*, 2001. **11**:31-48.
3. Carroll, J.M., and Rosson, M.B. Paradox of the Active User, in *Interfacing Thought: Cognitive Aspects of Human-Computer Interaction*, J.M. Carroll, ed. The MIT Press: Cambridge, MA, 1987, 80-111.
4. Chidamber, S.R., and Kemerer, C.F. A Metrics Suite for Object Oriented Design. *IEEE Trans. on Software Engineering*, 1994. **20**(6):476-493.
5. Crossen, A., Budzik, J., Warner, M., Birnbaum, L., and Hammond, K.J. XLibris: An Automated Library Research Assistant, in *Proc. of IUI'01* (Santa Fe, NM, 2001), 49-52.
6. Drummond, C., Ionescu, D., and Holte, R. A Learning Agent that Assists the Browsing of Software Libraries. *IEEE Trans. on Software Engineering*, 2000. **26**(12):1179-1196.
7. Fischer, G. Supporting Learning on Demand with Design Environments, in *International Conference on the Learning Sciences* (Evanston, IL, 1991), 165-172.
8. Fischer, G. User Modeling in Human-Computer Interaction. *User Modeling and User-Adapted Interaction*, 2001. **11**(1&2):65-86.
9. Fischer, G., et al. Seeding, Evolutionary Growth and Reseeding: The Incremental Development of Collaborative Design Environments, in *Coordination Theory and Collaboration Technology*, G. Olson, T. Malone, and J. Smith, eds. Lawrence Erlbaum: Mahwah, NJ, 2001, 447-472.
10. Fischer, G., Henninger, S., and Redmiles, D. Cognitive Tools for Locating and Comprehending Software Objects for Reuse, in *Proc. of 13th International Conference on Software Engineering* (Austin, TX, 1991), 318-328.
11. Fischer, G., Nakakoji, K., Ostwald, J., Stahl, G., and Sumner, T. Embedding Critics in Design Environments, in *Readings in Intelligent User Interfaces*, M.T. Maybury and W. Wahlster, eds. Morgan Kaufmann: San Francisco, CA, 1998, 537-559.
12. Fischer, G., and Ye, Y. Personalizing Delivered Information in a Software Reuse Environment, in *Proc. of 8th International Conference on User Modeling* (Sonthofen, Germany, 2001), 178-187.
13. Horvitz, E., Jacobs, A., and Hovel, D. Attention-Sensitive Alerting, in *Proc. of Conference on Uncertainty and Artificial Intelligence* (San Francisco, CA, 1999), 305-313.
14. Landauer, T.K., and Dumais, S.T. A Solution to Plato's Problem: The Latent Semantic Analysis Theory of Acquisition, Induction and Representation of Knowledge. *Psychological Review*, 1997. **104**(2):211-240.
15. Leake, D.B., Birnbaum, L., and Hammond, K.J. An Integrated Interface for Proactive, Experience-Based Design Support, in *Proc. of IUI'01* (Santa Fe, NM, 2001), 101-108.
16. Lieberman, H. Autonomous Interface Agents, in *Proc. of CHI'97* (Altanta, GA, 1997), 67-74.
17. Mili, A., Mili, R., and Mittermeir, R.T. A Survey of Software Reuse Libraries, in *Systematic Software Reuse*, W. Frakes, ed. Baltzer Science Publishers: Bussum, The Netherlands, 1998, 317-347.
18. Nakakoji, K., Yamamoto, Y., Suzuki, T., Takada, S., and Gross, M.D. From Critiquing to Representational Talkback: Computer Support for Revealing Features in Design. *Knowledge-Based Systems*, 1998. **11**(7-8):457-468.
19. Nardi, B.A., Miller, J.R., and Wright, D.J. Collaborative, Programmable Intelligent Agents. *Commun. ACM*, 1998. **41**(3):96-104.
20. Neal, L. Support for Software Design, Development and Reuse through an Example-Based Environment, in *Structure-Based Editors and Environments*, G. Szwillus and L. Neal, eds. Academic Press: San Diego, CA, 1996, 185-192.
21. Raymond, E.S., and Young, B. *The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary.* O'Reilly: Sebastopol, CA, 2001.
22. Rhodes, B.J., and Maes, P. Just-in-time Information Retrieval Agents. *IBM Systems Journal*, 2000. **39**(3&4):685-704.
23. Rich, C.H., and Waters, R.C. *The Programmer's Apprentice.* Addison-Wesley: Reading, MA, 1990.
24. Robbins, J.E., Hilbert, D.M., and Redmiles, D.F. Software Architecture Critics in Argo, in *Proc. of IUI'98* (San Francisco, CA, 1998), 141-144.
25. Schön, D.A. *The Reflective Practitioner: How Professionals Think in Action.* Basic Books: New York, 1983.
26. Ye, Y. *Supporting Component-Based Software Development with Active Component Repository Systems*, Ph.D. Dissertation, Department of Computer Science, University of Colorado, Boulder, CO, 2001
27. Zaremski, A.M., and Wing, J.M. Signature Matching: A Tool for Using Software Libraries. *ACM Trans. on Software Engineering and Methodology*, 1995. **4**(2):146-170.