Appeared in the Proceedings of 2007 International Conference on Principles and Practices of Programming in Java (PPPJ2007), ACM Press, Lisbon, Portugal, pp. 41-50, September, 2007.

Searching the Library and Asking the Peers: Learning to Use Java APIs on Demand

Yunwen Ye[1,3] Yasuhiro Yamamoto[2] Kumiyo Nakakoji[2,3] Yoshiyuki Nishinaka[3] Mitsuhiro Asada[3]

[1]L3D Center
Department of Computer Science
University of Colorado, Boulder, USA
+1-303-492-3547

[2]KID Lab
RCAST
University of Tokyo, Japan
+81-3-5452-5286

[3]SRA Key Technology Lab
3-12 Yotsuya, Shinjuku
Tokyo, Japan
+81-3-3357-9011

yunwen@colorado.edu {yxy, kumiyo}@kid.rcast.u-tokyo.ac.jp {nisinaka, m-asada}@sra.co.jp

ABSTRACT

The existence of large API libraries contributes significantly to the programming productivity and quality of Java programmers. The vast number of available library APIs, however, presents a learning challenge for Java programmers. Most Java programmers do not know all the APIs. Whenever their programming task requires API methods they do not yet know, they have to be able to find what they need and learn how to use them on demand. This paper describes a tool called *STeP_IN_Java* (a Socio-Technical Platform for In situ Networking of Java programmers) that helps Java programmers find APIs, and learn from both examples and experts how to use them on demand. STeP_IN_Java features a sophisticated yet easy-to-use search interface that enables programmers to conduct a personalized search and to progressively refine their search by limiting search scopes. Example programs are provided and embedded to assist programmers in using APIs. Furthermore, if a programmer still has questions about a particular API method, he or she can ask peer programmers. The STeP_IN_Java system automatically routes the question to a group of experts who are chosen based on two criteria: they have high expertise on the particular API method and they have a good social relationship with the programmer who is requesting the information.

Categories and Subject Descriptors

D.2.3 [**Software Engineering**]: Coding Tools and Techniques – *Object-oriented programming.* D.2.2 [**Software Engineering**]: Design Tools and Techniques – *Computer-aided software engineering, Software libraries.* H.5.3 [**Information Systems**]: Group and Organization Interfaces – *Computer-supported cooperative work*

General Terms

Design, Economics, Human Factors, Languages, Theory

Keywords

Java API, software reuse, expertise location, peer support

1. INTRODUCTION

The existence of large API libraries contributes significantly to the programming productivity and quality of Java programmers. The current Standard Edition of Java 1.6.0 (i.e., Java SE 6) has 3,777 classes and interfaces in its API library. As shown in Figure 1, the standard API library of Java SDK has grown nearly linearly over the past several years at the average rate of 356 classes and interfaces per year. In addition to the standard Java SDK APIs, many third-party API libraries from both proprietary companies and Open Source Software communities are being developed and becoming available for Java programmers.

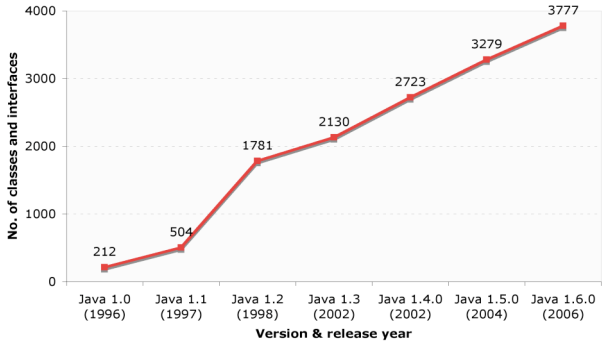

Figure 1: The growth of Java SDK API classes and interfaces

Given the sheer number of available APIs, few programmers, if any, know all of them. Whenever their programming tasks require API methods that they don't yet know, they have to be able to learn to use the API methods on demand. To learn to use a new API method on demand during a programming practice requires that a programmer quickly find the one that he or she needs, understand the specifics that are related to the current task, and integrate the API into his or her own programs.

Learning to use APIs on demand is different from other forms of learning in which the aim is to increase the general knowledge to prepare for potential future use. When the need for learning to use an unknown API method arises in a programming practice, the primary concern of the programmer is use, not the increase of general knowledge, because he or she has a pressing task to be accomplished [8]. Therefore, the process of learning to use a specific API on demand is highly personalized to the specific needs and existing knowledge of the programmer, and is tightly contextualized in the task and environment of the programmer.

The standard Javadoc documentation system is not very effective in supporting learning APIs on demand. First, its browsing system, based on the hierarchical structure of packages and classes, is not suited for finding unknown APIs. When a programmer is looking for an unknown API method during the work practices, he or she starts with a functionality requirement, not necessarily knowing what package or class might include the needed API method. Second, the documents are often insufficient for understanding the nitty-gritty details of the API methods. Many API methods are "inadequately documented—often by technical writers who aren't programmers and don't think like programmers" [15].

To complement the insufficiency of documents, programmers often engage in searching existing programs that use the desired API method to learn to use it. Finding a good, easy-to-understand example that illustrates the particular concerns that the programmer has is a time-consuming and tedious process [6].

In other cases, a programmer may enlist the help of peer programmers to ask specific questions about how to use the API method appropriately. This requires that the programmer know who has expertise on the particular API method [2]. Given the huge number of API methods, finding out who knows which API method is not an easy task. Furthermore, finding peer experts does not necessarily lead to acquiring their expertise. As knowledge resources, peer experts are different from other resources that are "things." "A thing is available at the bidding of the user—or could be—whereas a person formally becomes a skill resource only when he consents to do so, and he can also restrict time, place, and method as he chooses" [7]. Peer programmers, who are often constrained by their own programming tasks, must also be willing to share their precious expertise and time with the programmer who is asking for help.

This paper presents the STeP_IN_Java system that we have developed to provide *integrated*, *personalized,* and *contextualized* support for Java programmers to learn to use API methods on demand. It features a sophisticated yet easy-to-use search interface that enables programmers to conduct personalized searches based on functionality descriptions. Example programs are provided and embedded in documents to assist programmers in understanding how to use APIs in context. Furthermore, a programmer can post questions to an automatically selected group of peer experts who have expertise on the particular API method in question and are mostly likely willing to offer assistance in a timely manner. The system was designed and developed by instantiating the socio-technical framework described by Ye et al. [23].

2. SYSTEM OVERVIEW

The STeP_IN_Java system (Figure 2) is a web-based system, and users interact with it through web browsers. The system consists of four major subsystems. At the center is the *STeP_IN_Java Repository* subsystem, which stores the search index and documents of API libraries; examples; discussion archives; the technical profiles of programmers, which model the expertise they have about the indexed API libraries; and the social profiles of programmers, which model the social context by representing their relationships among peer programmers (Section 3). The *Profile Management* subsystem is used by programmers to initialize and to update their social and technical profiles. The *Search Engine* subsystem provides a personalized search for API methods, examples, and archived discussions. The *Peer Support* subsystem identifies and chooses experts to form an *ephemeral mailing list* through which a programmer can obtain help from peers.

To use the system, a Java programmer first has to register as a STeP_IN_Java user. After registration, the programmer needs to download a profiling client program (*Profiler*) and use that to create and upload his or her initial technical profile and social

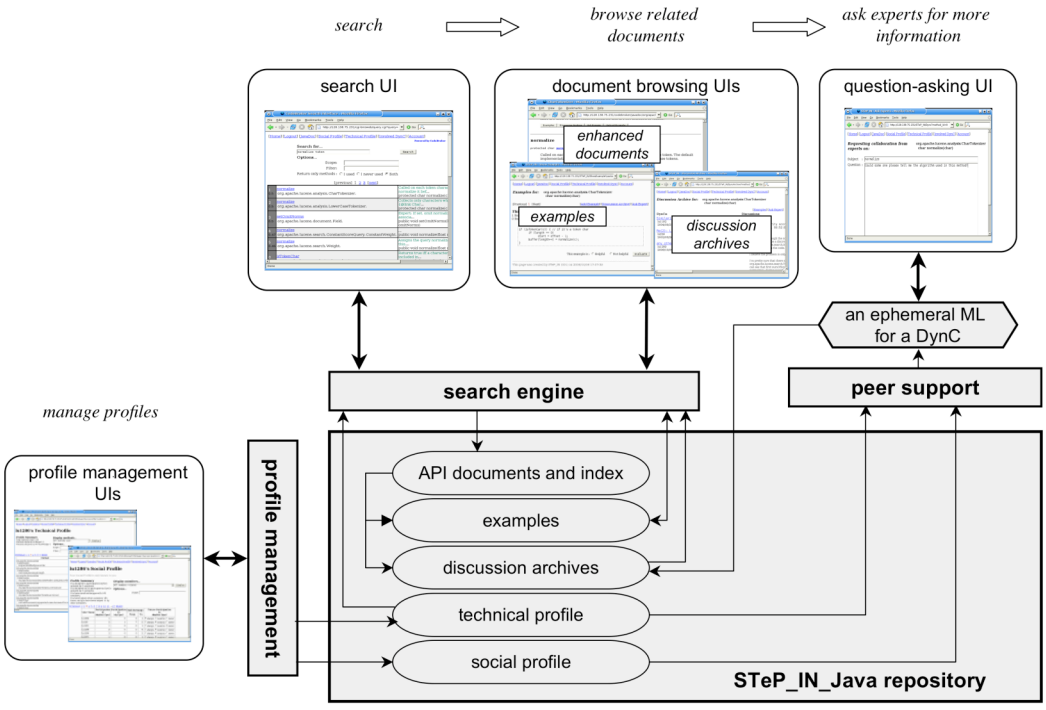

Figure 2: System overview

profile (details are provided in Sections 3.2 and 3.3, respectively).

A user who needs to learn to use APIs on demand interacts with STeP_IN_Java as follows after he or she has logged into the system.

1. The user types a natural language description of the functionality of the API method that he or she is looking for in the search interface. If too many search results are returned and the user cannot quickly find what is needed, he or she can use the *search-by-refinement* mechanism to personalize the search.
2. The user clicks on one of the returned results to bring up an enhanced Java document page. Each method in the Java document is enhanced with four embedded links: *Example*, *Discussion Archive*, *Ask Expert*, and *Upload Example* (see Figure 8 in Section 4.3 for an illustration).
3. By clicking on the *Example* link, the user opens an example page that shows code fragments that illustrate the use of the API method.
4. By clicking on the *Discussion Archive* link, which appears both in the Java document page and the example page, the user opens the page that displays archived past discussions on this particular method.
5. By clicking on the *Ask Expert* link, which appears in the Java document page, the example page, and the discussion archive page, the user gets a question composition window. He or she can then type a question about the method and submit it.
6. After the question is submitted, the system automatically selects a group of peer programmers whose technical profiles indicate they have expertise on the API method in question, and whose social profiles indicate that they have good social ties with the asker. The asker and the selected group of peer programmers (called helpers) become members of a dynamically created learning community (called DynC for short), and an ephemeral mailing list that consists of the DynC members is created.
7. Members of the DynC will receive the question posted by the asker through the ephemeral mailing list. The replies from the helpers are also sent to the same ephemeral mailing list.
8. If the asker deems that there is no more to discuss about the method, he or she should terminate the DynC by logging into STeP_IN_Java. Upon terminating the DynC, the asker is also required to evaluate the DynC as being "helpful" or "not helpful." The ephemeral mailing list associated with the DynC is also automatically terminated.

Questions and answers exchanged in the ephemeral mailing list are archived in the STeP_IN repository and linked to the API method around which the discussion is centered.

3. REPOSITORY AND PROFILE MANAGEMENT

The repository of the STeP_IN_Java system contains the search index for Java API methods, documents of each method, examples of method usage, accumulated discussions, and technical profiles and social profiles of programmers.

3.1 Indexes and Documents

STeP_IN_Java treats each Java API method as an indexing and search unit. It uses the probability-based free-text information retrieval technique [16] to index each API method based on its name and text-based javadoc description [22].

From the source code of the API libraries, STeP_IN_Java uses the Doclet API to generate the documents and search indexes to populate its repository.

3.2 Technical Profiles

In the STeP_IN_Java system, each user has a technical profile to represent his or her existing expertise about the indexed API methods. The technical profile of a programmer includes four sets of API methods:

(1) The set of *Used Methods* includes those methods that the programmer has used in his or her own programs.

(2) The set of *Confirmed Known Methods* includes those methods for which the programmer has demonstrated expertise through replying to questions on these methods asked by other programmers.

(3) The set of *Claimed Known Methods* includes those methods that the programmer claims he or she knows.

(4) The set of *Not Interested Methods* includes those methods about which the programmer does not want to share expertise with other peers.

The sets of (1) and (2) are objective approximations of the expertise that a programmer has and are automatically maintained and updated by the system. The sets of (3) and (4) are subjective choices that the programmer has made by using the *Profile Management* subsystem.

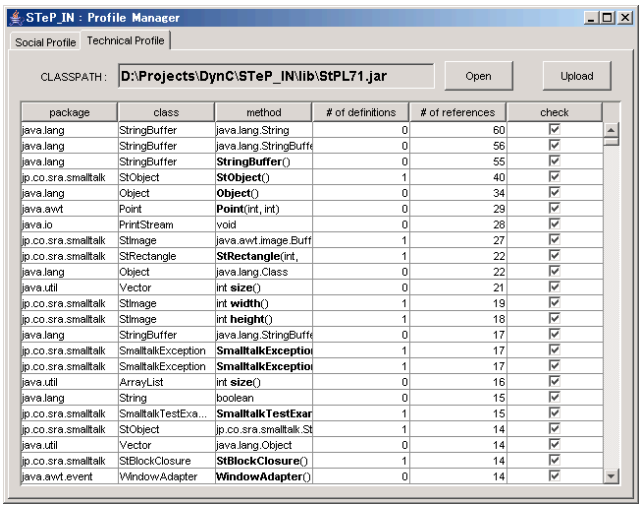

Figure 3: Technical profile initialization

A programmer needs to initialize his or her technical profile by using the *Profiler* when registering into the system. As shown in Figure 3, the programmer needs to specify in the *Profiler* the *CLASSPATH* to the Java programs that he or she has written so far. The *Profiler* parses the programs and extracts all the method references. The extracted references of the methods that are not indexed in the STeP_IN_Java repository are discarded, and the remaining references of the indexed methods are displayed, with the total number of references of each method in all programs. The method names and reference numbers are then uploaded to the STeP_IN_Java system and become elements in the *Used Methods* of the programmer's technical profile. Programmers can uncheck the *check* field if they do not want a method to be included in their initial technical profiles as a known method.

Another source for initializing the technical profile of a programmer is his or her mailbox because programmers routinely

use emails to communicate with peer programmers. The *Profiler* is therefore also able to extract what programmers know about the indexed API methods by analyzing their mailboxes. Because mail messages are text, we cannot use Java parsers to extract method names and their reference numbers. Instead, the STeP_IN_Java *Profiler* uses the following heuristics to extract the known methods:

Step 1: Remove quoted parts in an email message by recognizing the conventions used by major mail programs.

Step 2: Split the message into *words* along white spaces (space, tab, and carriage return).

Step 3: For each *word*, determine if it is a part of Java code fragments based on whether it contains, or is followed by such telltale characters as "`()=.+-`".

Step 3.1: If the *word* is determined to be part of a possible Java code fragment, check whether it agrees with Java method call syntax.

Step 3.2: If the *word* agrees with the Java method call syntax, extract the method name, and class and package names if they are available.

Step 3.3: Look up the method name (with class name and package name, if available) in the repository of indexed methods. If the method name is unique, it is regarded as one count of usage of the method by the programmer. If the method name is not unique (especially when class name and package name are not available), it is discarded.

The extracted numbers of usage of the index methods are then added to the *Used Methods* of the programmer's technical profile.

Figure 4: Technical profile management

Most of the technical profile of a programmer is therefore automatically created and maintained by the system. However, a programmer can, if desired, update his or her technical profile through the *Profile Management* subsystem (Figure 4) by checking the *Expert* button or the *No Knowledge* button at any time. If the programmer checks the *Expert* button on an API method, the method is added to the *Known Methods* set of his or her technical profile. If the programmer checks the *No Knowledge* button, the method is added to the *Not Interested Methods* set of his or her technical profile. When an API method is added to the *Not Interested Methods* set of a programmer's technical profile, no matter whether the programmer actually knows the method, he or she will not receive any questions about the method. This mechanism grants programmers the control to exclude themselves from answering questions on selected methods for whatever reasons. For example, a programmer might not want to answer questions on API methods that he or she thinks are too low level to match his or her expertise, wanting to focus on answering more interesting and challenging questions. Or, a programmer might get bored of answering questions about the same API method again and again.

3.3 Social Profiles

A programmer's social profile represents his or her existing social relationships with other STeP_IN_Java users that resulted from their previous social interactions. A social profile defines four types of social relationships: *exclude, include, help,* and *email*. The first two are claimed relationships that are explicitly specified by the programmer, and the latter two are factual relationships that record the history of interactions with other programmers. All four kinds of relationships are unidirectional.

- **exclude<P, Q>:** This means that *P* does not want to answer any questions asked by *Q*. In other words, *P* does not want to participate in any DynC initiated by *Q* on any API method, regardless of whether *P* has expertise in it. This choice is known only to *P*; no one else, including *Q*, knows it. Because relationships are unidirectional, *exclude<P, Q>* does not mean *exclude<Q, P>* because the perception of social relationships is subjective and not necessary mutually equal.
- **include<P, Q>:** This means that *P* is always willing to help *Q* if *P* has any level of expertise on any API method that *Q* wants to learn. Similarly, this relationship can be seen and edited only by *P*, and *include<P, Q>* does not mean *include<Q, P>*. It could even be possible that both *include<P, Q>* and *exclude<Q, P>* exist at the same time, meaning that *P* is always willing to help *Q*, while *Q* does not want to share expertise with *P*. However, *exclude<P, Q>* and *include<P, Q>* are mutually exclusive.
- **help<P, Q>:** This represents the number of times that *P* has helped *Q* within the STeP_IN_Java system. If *P* replies to the question posted by *Q* in the DynC initiated by *Q*, STeP_IN_Java adds one count to *help<P, Q>*.
- **email<P, Q>:** This represents the number of emails that *P* has sent to *Q* outside of the STeP_IN_Java system.

The *email* relationship constitutes the initial value of the social profile, and is added to his or her social profile when a user registers in the STeP_IN_Java system with *Profiler*. After the user provides the path to his or her mailbox, the STeP_IN_Java *Profiler* extracts the email addresses of those who have sent emails to him or her and the number of the emails sent, and uploads the senders and the number of emails (but no other information) to the STeP_IN_Java system. Similarly, users can uncheck the item if they do not want to include in their social profiles any email exchange information with a particular person.

Users can also update their social profiles through the *Profile Management* subsystem. A social profile is visible and editable only by the user. Figure 5 shows the interface for updating the social profile of user *lu1286*. The second column, *Participation in his/her DynC*, shows how many times *lu1286* has helped a specific programmer (whose name is shown in the first column) by participating in DynCs initiated by that programmer. For example, row 1 shows that *lu1286* has helped *lu1259* once, and this is modeled in *help<lu1286, lu1259>*. The third column, *Participation in My DynC*, shows how many times a specific programmer has tried to help by participating in the DynCs initiated by *lu1286*. For example, as shown in row 1, *lu1259* has never helped *lu1286*, and this is modeled in *help<lu1259, lu1286>*. Both above numbers as well as the numbers of email

exchanges are facts that cannot be changed. However, *lu1286* can specify whether he or she is willing to continue providing help to programmer *lu1259* in the future by choosing one of the three options in the last column, *Future participation in his/her DynC*. For example, in Figure 5, *lu1286* chooses *always* in the row for *lu1259*, and *include<lu1286, lu1259>* is added to *lu1286*'s social profile. Note that *lu1286* choosing *never* in the row for *lu1261*, adding *exclude<lu1286, lu1261>* to *lu1286*'s social profile.

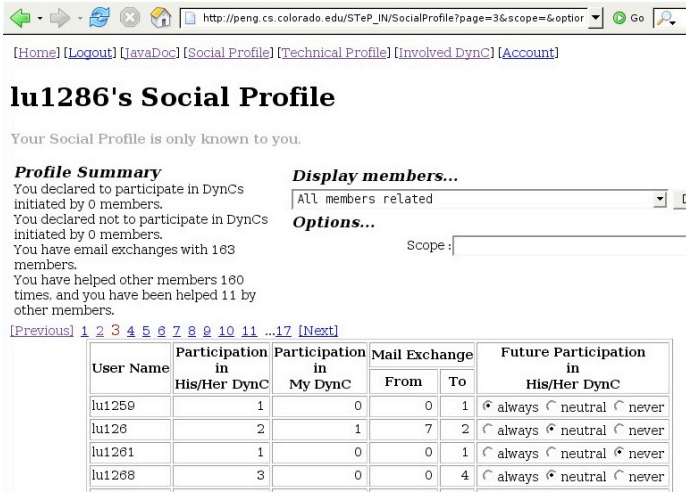

Figure 5: Social profile management

4. SEARCH ENGINE

User queries are written in natural language. Retrieval results are returned based on the similarity between user queries and the documents of each method in the repository (Figure 6). Based on the assumption that terms are distributed differently in relevant and irrelevant documents, the probability-based information retrieval technique that is adopted by the STeP_IN_Java system computes the similarity between a query and a document by assigning appropriate weights to terms in the document collection, and returns a rank-ordered list of indexed documents that best match the query [16, 21].

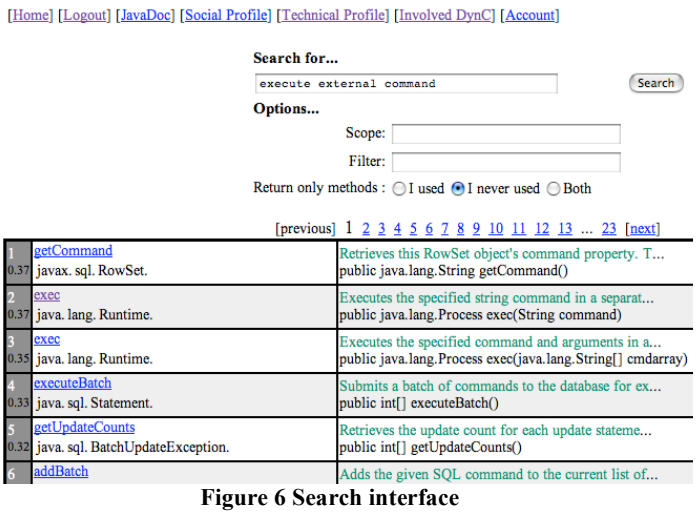

Figure 6 Search interface

4.1 Personalized Search

STeP_IN_Java supports personalized searches. As mentioned in Section 3, users' technical profiles represent what methods they have known by analyzing the Java programs they have written and emails they have sent. In STeP_IN_Java, users can choose to limit their search range to all the methods that they have used, or to limit their search range to all the methods that they have never used. The former mechanism is meant to support the search for those methods that users vaguely know and have used but cannot remember the details; the latter mechanism is meant to support the search for completely unknown methods.

4.2 Iterative Search by Refinement

Search-by-refinement is also supported. Information search is seldom a one-shot action, due to the difficulty of formulating a perfect query when the search object is not clearly known and well-defined in advance [12]. Information retrieval systems can, at best, retrieve information that matches the queries submitted by a user, and the retrieved information may not necessarily match the user's real intentions, which may not be fully articulated in the query. Search-by-refinement [20] is a process that allows users to incrementally improve their queries after they have examined the initial retrieval results.

Java API libraries are separated into packages and classes. For most programming tasks, only a small portion of the packages and classes are needed. If the search is limited to those packages or classes that are relevant to the current task, search efficiency will be greatly improved. In the search results returned by the search engine of STeP_IN_Java, each method name is accompanied by the full class name to which the method belongs. As the user moves the mouse cursor over the package name, or any subpackage name, or the class name, the name will be highlighted; if the user then clicks the mouse, a small window will appear below the full class name (Figure 7). The users can click either the *+Scope* or *–Filter* option.

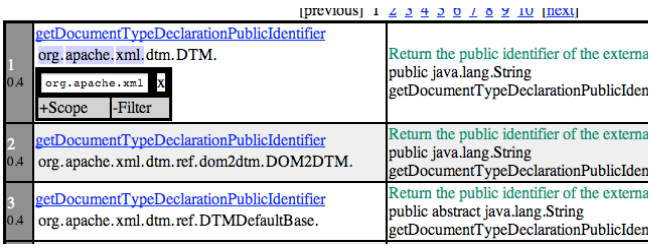

Figure 7: Query-by-reformulation in context

If *+Scope* is clicked, the search results will be limited to the specified package or class. For example, in Figure 7, only methods from the packages that start with *org.apache.xml* will be returned. Conversely, if *–Filter* is clicked, all methods from the packages that start with *org.apache.xml* will be removed from the search results. For the same search task of looking for the API of executing an external command as shown in Figure 6, if the programmer formulates his or her query as "external command" instead of "execute external command" as shown in Figure 6, the *exec* method that matches the real intention of the programmer will not be returned in the first page of the retrieval results. The first several methods are all from *org.apache.xml*, as shown in Figure 7. By examining the results, the programmer can soon realize that what he or she is looking for probably won't be in *org.apache.xml*. The programmer can then choose to filter this package out, and the *exec* method will pop up to the first page of the retrieval results.

Similar search range specification can be included in the initial search if users write *org.apache.xml* in the *Filter* field of the search interface (Figure 6). However, for most programmers who

are not very familiar with the library, it is much each easier to specify the range after they have seen the initial search results.

4.3 Enhanced Java API Documents

If the programmer decides to further examine an API method from the retrieval results, he or she can click the method name. This will bring up the documents for that API method.

As mentioned before, STeP_IN_Java extends the standard Java API documents with four added buttons: *Example, Discussion Archive, Ask Expert,* and *Upload Example* (Figure 8).

If the document does not provide enough information for the programmer to learn to use the API, he or she can click the *Example* button to take a look at code fragments that illustrate the use of the API method (Figure 9).

Further information can be found by clicking the *Discussion Archive* button (from either Figure 8 or Figure 9), which displays archived previous discussions about the API method (Figure 10).

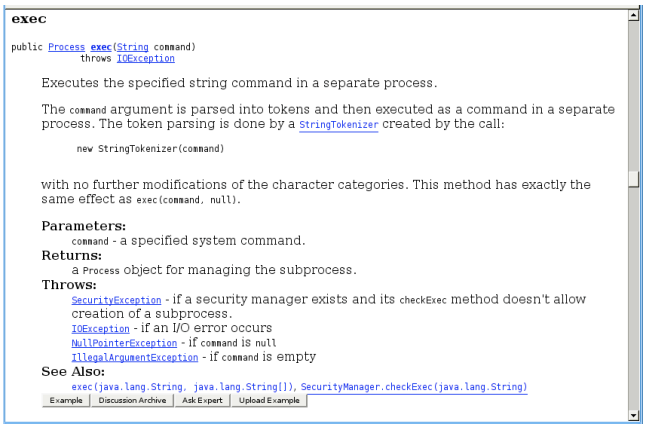

Figure 8: Extended Java API documents

Figure 9: Example code

Figure 10: Discussion archive

5. ASKING EXPERTS

When reading the document, examples, and discussion archive is not enough for the programmer to understand how to use the API method, he or she can seek help from peer programmers. Clicking the *Ask Expert* button—which appears in the document page (Figure 8), the example page (Figure 9) and the discussion archive page (Figure 10)—opens a window for posting a question (Figure 11). For the sake of brevity, we will use *Bob* to denote the programmer who is currently asking a question, and *mtd* to denote the API method that is being asked about.

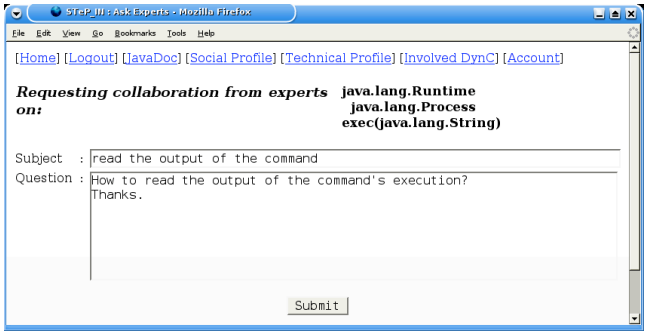

Figure 11: Asking a question

As soon as *Bob* submits the question, the STeP_IN_Java system automatically sends the question to a group of selected peer experts. The group is a dynamically created learning community initiated by *Bob* on the topic of method *mtd* and this group, denoted as *DynC(Bob, mtd)*, has members determined through the following steps.

5.1 Identifying Expert Peers

The first step identifies those peer programmers who have expertise on *mtd* by examining each peer programmer's technical profile. If the *Used Methods*, *Claimed Known Methods*, or *Confirmed Known Methods* in a peer programmer's technical profile includes *mtd*, the peer programmer is identified as an expert, and is added to the *List of Candidate Experts*.

5.2 Removing Unwilling Expert Peers

Having expertise is only a necessary condition for expertise sharing. The programmers who hold the expertise must also be willing to share their expertise with other programmers. If the programmer is unwilling to share his or her expertise, sending the question to the programmer will not help the asker obtain the needed learning help. Additionally, a programmer who is unwilling to help but still receives the question will be unnecessarily annoyed and interrupted from his or her own programming task.

To avoid this unnecessary annoyance and interruption incurred on unwilling programmers, the STeP_IN_Java system removes two types of peer experts who have explicitly indicated unwillingness.

For each programmer in the *List of Candidate Experts*, if the *Not Interested Methods* of the programmer's technical profile includes *mtd*, the programmer is removed from the *List of Candidate Experts* because the programmer has already explicitly declared that he or she is not interested in answering questions about *mtd*.

The second reason that a peer programmer might not be willing to help is a factor of individual relationships. An arduous relationship between the source of expertise and the recipient of expertise is a well-recognized impediment to expertise sharing [3].

STeP_IN_Java therefore removes from the *List of Candidate Experts* those peer programmers who have explicitly indicated their unwillingness to help the asker. In other words, for each member *K* in the *List of Candidate Experts*, if *K* has explicitly specified that he or she does not want to participate in helping *Bob* (i.e., *exclude<K, Bob>* exists), *K* is removed from the *List of Candidate Experts* for *Bob*.

5.3 Selecting Expert Peers Mostly Likely to Help

All the remaining members of the *List of Candidate Experts* don't have explicit personal preferences of not answering questions on *mtd*, or of not helping *Bob*. However, this does not mean that all of them will be equally willing to help *Bob*. Empirical studies have shown that existing social ties improve the motivation of the helper and the quality of the helping act [3]. To further improve the effectiveness of obtaining expertise from peer programmers, the STeP_IN_Java system conducts another selection round by considering the social ties that exist between the candidate experts and *Bob*, and it selects peer experts who are most likely willing to help *Bob*.

First, the system considers whether a candidate expert has explicitly indicated his or her willingness to help *Bob*. For each member *K* in the remaining *List of Candidate Experts*, if *include<K, Bob>* exists, which means that *K* has declared he or she will participate in DynCs initiated by *Bob*, *K* is then selected as a member of *DynC(Bob, mtd)*.

Second, the system selects those expert candidates who have been helped by *Bob* before. In other words, for each member *K* in the remaining *List of Candidate Experts*, if *help<Bob, K>* exists, which means that *Bob* has helped *K* before, *K* is then selected as a member of *DynC(Bob, mtd)*. Because *K* has been the recipient of the help provided by *Bob* in the past, it is highly likely that *K* is willing to reciprocate the favor this time.

Third, the system selects those expert candidates who have received more help in general. For each member *K* in the remaining *List of Candidate Experts*, based on the *help* relation in *K*'s social profile, if the number of helps that *K* has received from others is greater than the number of helps that *K* has offered to others, namely,

$$\sum help<P, K> > \sum help<K, P>$$

where *P* is any registered member of STeP_IN_Java,

K is then selected as a member of *DynC(Bob, mtd)*. Although *K* and *Bob* do not have a direct helping relationship, due to the social norm of generalized reciprocity that regulates social interactions among group members, *K* has social obligations to return favors that he or she has received from the group in the past by offering help to other members [14].

The above selection rules rely on the captured *help* relationships that have resulted from the interactions among programmers taking place inside the STeP_IN_Java system. However, when the system is initially deployed, or when a new member registers to the system, there will not be enough historical data to make the above selections. As a way of jump-starting, if the above rules fail to select any peer experts, the STeP_IN_Java system selects peer experts by utilizing existing social ties that are reflected in the history of email exchange. Namely, for each member *K* in the *List of Candidate Experts*, if *email<K, Bob>* exists, which means that *K* has sent emails to *Bob*, then *K* is selected as a member of *DynC(Bob, mtd)*. The fact that *K* has sent emails to *Bob* indicates the possibility that *K* knows *Bob* to a certain degree, which further implies that *K* might be willing to help *Bob*.

5.4 Creating an Ephemeral Mailing List

The finally selected experts become the members of *DynC(Bob, mtd)*, and an ephemeral mailing list that consists of the selected members is dynamically created. Through the ephemeral mailing list, the *DynC(Bob, mtd)* members receive the question posted by *Bob* on the API method *mtd*.

All the replies from *DynC(Bob, mtd)* members are also sent to the ephemeral mailing list associated with *DynC(Bob, mtd)*. When *Bob* feels the discussion is sufficient, he should close *DynC(Bob, mtd)* after evaluating the DynC as "helpful" or "not helpful." Or, if no message is exchanged for a predetermined period of time, *DynC(Bob, mtd)* will automatically be closed by the system. Once a DynC is closed, its associated ephemeral mail list is discontinued as well.

All the discussions that took place in the ephemeral mailing list are archived in the STeP_IN_Java system and are linked to the API method. Other programmers are still able to reap the benefits of expertise sharing by browsing the discussion archive (Figure 10), even though they were not directly involved in the original discussion.

5.5 Updating Profiles

As programmers ask and answer questions in the STeP_IN_Java system, their social relationships with other programmers change, and such changes are captured and reflected in the updating of their technical and social profiles.

If asker *Bob* evaluates *DynC(Bob, mtd)* that he initiated as "helpful," any member *K* of *DynC(Bob, mtd)* who has sent an email to *DynC(Bob, mtd)* is regarded as a known expert on method *mtd*. Namely, the *Known Methods* of *K*'s technical profile will now include *mtd*. At the same time, the system records the fact that *K* has helped *Bob* once, and increases the value of *help<K, Bob>* by one.

In addition to the automatic update of profiles, the experts who receive the request for help are also offered an opportunity to change personal preferences in their profiles. The question email that each member of *DynC(Bob, mtd)* receives is embedded with two personalized links to his or her profile management interface. One link takes the member to his or her technical profile management interface (Figure 4) with the method *mtd* shown. By clicking the *I don't know* button at the row for *mtd*, the member will no longer receive any questions that are related to *mtd*.

The other link takes the member to his or her social profile management interface (Figure 5) with the name of *Bob* shown. By clicking the button *never* for the row of *Bob*, the expert will no longer receive any questions that are asked by *Bob*.

The two embedded links are meant to ease the burden of maintaining updated individual profiles. A programmer who feels that he or she is involved in an unwanted DynC can react immediately and take action by using the embedded links. This removes the need for the tedious and time-consuming initial setup of individual profiles. It also makes it easier for programmers to make decisions by reacting to a concrete situation instead of thinking abstractly when they are asked to maintain their profiles separately.

6. SYSTEM EVALUATION

Evaluating a system such as STeP_IN_Java is very challenging because it deals with human preferences and social perceptions. A longitudinal study of its use in real development environments is required to thoroughly evaluate the system. We do not yet have real usage data to report on the overall effectiveness of the system. However, we have conducted preliminary evaluations of the key techniques: the API method search mechanism and the automatic identification of expert programmers on a particular API method.

6.1 Evaluating the Search Mechanism

Information retrieval systems are conventionally evaluated by *recall* and *precision* [18]. Recall is the proportion of relevant material actually retrieved in answer to a search query, and precision is the proportion of retrieved material that is actually relevant. Figure 12 shows the recall-precision curve for the results of executing 19 queries. For more details about the evaluation of the search mechanism, please see the evaluation of the *CodeBroker* system [22], whose search engine is reused in the STeP_IN_Java system.

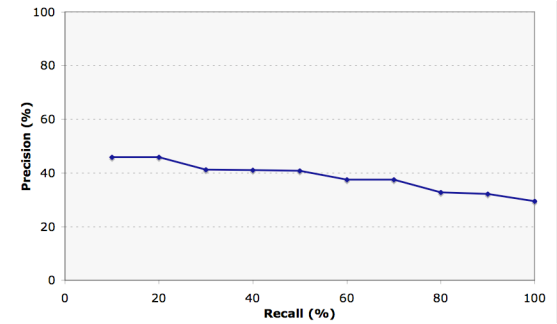

Figure 12: Precision-recall curve

6.2 Evaluating the Expert Finding and Selecting Mechanism

We have conducted a simulation study to investigate how STeP_IN_Java is able to identify and select the "right" experts to receive questions asked by a Java programmer.

6.2.1 Data Set

The data we used for the evaluation study are the Java API library of the Apache Lucene system (http://lucene.apache.org/) and the emails posted to the *java-user@lucene.apache.org* mailing list, which is used by programmers who use the Lucene Java API library.

The messages sent between 2001 and 2005 to the mailing list were used as the base data. We then simulated how DynCs would be formed for the questions posted during 2006, and compared the results with the actual conversation threads carried out in the mailing list in 2006.

A total of 2,291 members posted messages to the mailing list. During the period between 2001 and 2005, which we used as the base data, there were 17,942 messages with 4,616 threads identified.

6.2.2 Procedure

The simulation study was conducted as follows.

(1) We indexed the Lucene Java API library and populated the STeP_IN_Java repository with indexes and enhanced Java documents from its source code.

(2) We wrote a Python script to process the mailing list archive and identify 2,291 members who sent emails to the mailing list. Individual members who use different email addresses were identified through a set of heuristics and were visually verified before merging them into one identity. The 2,291 members were then registered as STeP_IN_Java users, with pseudo usernames such as U0001, U0002, ..., U2291.

(3) We extracted all the emails that each member had sent to the mailing list to construct a mailbox for him or her. From the constructed mailbox, we created each member's technical profile.

(4) We divided the mailing list archive into conversation threads. For each thread, we identified the member (e.g., *A*) who sent the first message to initiate the discussion. We then identified those members who replied to the thread (e.g., *B, C, D*) and presumed the existence of *help* relationships: *help<B, A>*, *help<C, A>* and *help<D, A>*.

(5) We then simulated communications that took place in 2006. For an actual thread that started during 2006, if the thread was asking about a certain Lucene Java API method, we logged into STeP_IN_Java as the initiator of the thread, and clicked the *Ask Expert* button from the method's document to ask the same question. STeP_IN_Java then created a DynC for the thread-initiator about the API method. The group of members that was selected by the STeP_IN_Java system to receive the question was compared with the set of users who actually replied to the thread in the actual mailing list.

6.2.3 Results

Table 1 shows the results of the experiment. We listed the first 20 API methods about which questions were posted in the mailing list in 2006. Table 1 shows the subject of the question (Column 2); the initiator of the thread (Column 3), who is the asker in the STeP_IN_Java system; the programmers who actually posted replies in the java-user@lucene.apache.org mailing list (Column 4); and the DynC members selected by the STeP_IN_Java system (Column 5). Shaded usernames in both Columns 4 and 5 show those who actually replied to the thread in the mailing list and were also selected by STeP_IN_Java.

Table 1: The experiment results

Thread No.	Subject	Asker	Actual repliers in the Lucene ML				DynC members identifed in SIJ				
1	*My first question in 2006 :D*	U1293	U1156				U0126	U0256	U0065	U1300	U1186
2	*numDocs() after undeleteAll()*	U1671	U1162				U0568	U1102	U1162	U1978	U1086
3	*Lock obtain timed out + IndexSearcher*	U1605	U0065	U1162			U0126	U0065	U0703	U0561	U1185
4	*Generating phrase queries from term queries*	U0390	U1162	U0953	U0549	U1286	U0292	U1558	U0953	U1787	U0775
5	*How do I get a count of all search results inside of my content?*	U1858	U1286				U0953	U0292	U0126	U1799	U1286
6	*What's the differents between QueryParser and Query*	U1856	U0126				U1034	U1293	U1329	U1307	U0070
7	*Range queries*	U1871	U0126	U1291	U1286	U1754	U1117	U0321	U1070	U1162	U0126
8	*How does the lucene normalize the score?*	U1214	U1276	U1823	U1286	U1162	U1823	U1276			
9	*Sorting by Score*	U1757	U0499	U1286			U0126	U1286	U1034	U1184	U0776
10	*index merging*	U1409	U1162	U1246	U1772		U1246	U1949	U2161	U1845	U1622
11	*How to get mapping of query terms to number of their occurrences in a doc?*	U1837	U1286	U0126	U0065		U1817	U0720	U1286	U1162	U1795
12	*query formulation*	U1606	U1162				U2061	U1162	U1286	U1777	
13	*How can I get a term's frequency?*	U1928	U1772	U0849			U1198	U1144	U1356	U0628	U0553
14	*Efficiently updating indexed documents*	U1926	U1162				U1952	U1162	U1074	U1456	U1978
15	*Help on Similarity*	U1777	U1286				U1499	U0789	U0873	U0325	U0292
16	*sumOfSquaredWeights for lengthNorm*	U1777	U1286				U0126	U1286	U1192	U1149	U1230
17	*Get only count*	U1792	U0390	U1162	U0953		U0531	U0100	U1192	U1684	U1149
18	*Reading stop word from a file!*	U1778	U1978				U1978				
19	*Changing ranking*	U1884	U0065	U1582	U1286		U2055	U1286	U1963	U1162	U2108
20	*writeChars method in IndexOutput*	U2009	U1162				U1162				

In seven cases (Thread Nos. 2, 5, 12, 14, 16, 18, and 20), all the actual repliers in the mailing list have been selected by the STeP_IN_Java system as DynC members. In eight cases (Thread Nos. 3, 4, 7, 8, 9, 10, 11, and 19), there is partial matching. Five

cases (Thread Nos. 1, 6, 13, 15, and 17) have no matching. This means that if the Lucene Java API user community had been using the STeP_IN_Java system, 15 askers would have been able to get responses through DynCs, provided the technical and social profiles we simulated remain the same.

Based on the results, we are optimistic that STeP_IN_Java would perform well in real use because we can expect more precise profiles. The technical profiles used in the experiment were created from the email messages. In real use, Java programs written by programmers will be available for the creation of technical profiles, and we will have more complete and accurate technical profiles that reflect the expertise of programmers. However, the experiment also points to a number of issues for further exploration, which will be discussed next.

7. DISCUSSION

By exploring the five cases of no matching in the simulation study, we found that some of the actual repliers are programmers who develop the Lucene Java API library. Their expertise was not fully accounted for in their technical profiles because some of them only recently become active in this user support mailing list. This points to the need to modify the definition of expertise levels when library developers and application developers are mixed together.

Some members are eager helpers [19] who are very motivated to help others. For example, in Thread No. 6, U0126, who had no previous direct interactions with U1856, offered his help. A further examination revealed that U0126 is clearly an eager helper because he has helped other users 552 times and received help only 61 times. In real use, when the question is sent only to the DynC members, some of the DynC members might have answered; or the eager helper U0126 could declare an *include* relationship with all users and gets selected whenever he has matching expertise. However, there remains the possibility that a question posted to DynC misses answers provided by an eager helper who was not selected. An escalation mechanism is being considered for future addition: When no answers are provided in a DynC, the system might need to expand the selection of DynC members by including all experts and finally maybe the whole community.

One may argue, why not ask all the experts or all the members in the first place through mailing lists to which all members subscribe? Mailing lists certainly have their advantages. In mailing lists, or their variation (e.g., bulletin board systems), anyone can volunteer expertise whenever he or she feels like it. The mailing list subscribers identify themselves as experts upon reading a question. However, the asker has no way of controlling the quality of the answers or pushing for an answer, and has to wait for the experts to show up.

At the same time, other subscribers who have neither expertise nor interest in helping the particular asker on the question have to spend some time dealing with all the emails. In particular, when such a mutual helping system as STeP_IN_Java is deployed in a corporate setting, the time disinterested users spend to deal with unwanted questions takes away from the limited time they can otherwise use for productive programming work.

From the asker's perspective, if every question asked would always go to all members of the mailing list, the asker risks giving those colleagues the impression that he or she is rather ignorant and incompetent [5]. In the STeP_IN_Java system, the questions are routed to different sets of peers. More important, the receivers of the questions already have affinitive social relationships with the asker, which provides a psychological safety net for asking questions [3]. From the perspective of the experts who receive questions, they have fewer questions to answer and are interrupted less frequently.

The other benefit offered by mailing lists is the opportunity for passive learning, meaning those who are not directly involved in the questioning and answering can learn by reading the messages or obtain an awareness of what is going on. This benefit is still retained in the STeP_IN_Java system because all messages are archived and linked to the API method.

8. RELATED WORK

The STeP_IN_Java system is related to a number of systems that have been developed to help programmers acquire external expertise for their programming task.

Support for searching library methods has been a major research theme in software reuse [12, 21, 22]. However, most of the reuse research has focused on devising and evaluating different kinds of indexing and searching methods. The search support of the STeP_IN_Java system differs from previous reuse systems in its personalization and incremental refinement of searching activities.

Recently, a few systems have been developed specifically to support searching the increasingly large Java API libraries. The CodeBroker system [21] that we previously developed helps Java programmers learn to use unknown Java API methods without explicit search actions. The system monitors programming activities in the program editor, infers what might be needed by the programmer, and then automatically recommends library API methods that are relevant to the current programming task. This automatic recommendation of Java API methods is further explored by the RASCAL system [10]. The Hipikat system [4] takes a similar approach and supports incremental development by recommending code as well as development documents from the historical data of a software project.

Several systems, such as Prospector [9], XSnippet [17], and Strathcona [17] have been developed to find code fragments that illustrate the use of Java API to facilitate learning. Java programmers can also search many Open Source Software systems to find code examples by using websites such as koders.com, docjar.com, and google.com/codesearch.

The task of finding peer experts to help programmers has been explored in the past, although none of the systems target helping programmers to learn Java API methods. Answer Garden [1] is an early attempt that routes questions about Unix to experts based on predefined expertise profiles, and accumulates answers from experts. The Expertise Recommender system [11] mines configuration management logs to identify experts and recommends experts based on organizational relations to support software maintainers. The approach of identifying experts from project history was further improved and validated in the Expertise Browser system [13]. Our approach differs from these by considering social factors that affect expertise sharing and acquisition.

9. SUMMARY

The effective use of library API methods is fundamental to Java programming. Learning to use the fast-growing library API remains a challenge for most Java programmers: programmers

often need to quickly find the API method that suits their tasks, and to understand its subtlety, which is often undocumented. We have developed the STeP_IN_Java system to support Java programmers in finding and learning to use API methods on demand, with the following distinctive features:

(1) The system provides a comprehensive learning support that ranges from searching, examples, discussion archives, and asking peer experts.

(2) Search is personalized to the background knowledge of the programmer and his or her immediate task context.

(3) Peer experts are automatically found when the programmer needs to obtain help from peers. The finding of peer experts is contextualized to the programmer's current task and surrounding social relationships. If the same programmer asks questions on different API methods, each group of found experts will be different because peers who have expertise on the methods are different. If different programmers ask a question on the same API method, different groups of experts will be found because different programmers have different social relationships with others. If the same programmer asks a question about the same method at different times, different groups of experts might be found because social relationships and individual expertise change as programmers interact with each other.

The technical evaluation of the search and expert finding mechanisms of the system has shown promise. Our future work includes conducting longitudinal evaluations of the system through its use in a real context to understand the effects of the individual preferences that programmers make to their technical profiles and social profiles.

Acknowledgments. This research was partially supported by MEXT Open Competition for the Development of Innovation Technology, No. 15103, and MEXT Grant-in-Aid for Exploratory Research, 17650038, 2005.

10. REFERENCES

[1] Ackerman, M.S. and T.W. Malone, Answer Garden: A Tool for Growing Organizational Memory, in *Proceedings of the ACM Conference on Office Information Systems*. 1990: Cambridge MA. p. 31-39.

[2] Berlin, L.M., Beyond Program Understanding: A Look at Programming Expertise in Industry, in *Empirical Studies of Programmers: Fifth Workshop*, C.R. Cook, J.C. Scholtz, and J.C. Spohrer, Editors. 1993, Palo Alto, CA: Ablex Publishing Corporation. p. 6-25.

[3] Cross, R. and S.P. Borgatti, The Ties That Share: Relational Characteristics That Facilitate Information Seeking, in *Social Capital and Information Technology*, M. Huysman and V. Wulf, Editors. 2004, Cambridge, MA: The MIT Press. p. 137-161.

[4] Davor C. Cubranic and G.C. Murphy, Hipikat: Recommending Pertinent Software Development Artifacts, in *Proceedings of the 25th International Conference on Software Engineering (ICSE03)*. 2003: Portland, OR. p. 408-418.

[5] Flammer, A., Towards a Theory of Question Asking. *Psychological Research*, 1981. **43**: p. 407-420.

[6] Holmes, R. and G.C. Murphy, Using Structural Context to Recommend Source Code Examples, in *Proceedings of 27th International Conference on Software Engineering*. 2005. p. 117-125.

[7] Illich, I., *Deschooling Society*. 1971, New York: Harper and Row.

[8] Lange, B.M. and T.G. Moher. Some Strategies of Reuse in an Object-oriented Programming Environment, in *Proceedings of Human Factors in Computing Systems*. 1989. Austin, TX: ACM Press.

[9] Mandelin, D., et al., Jungloid Mining: Helping to Navigate the API Jungle, in *Proceedings of 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2005: Chicago, IL. p. 48-61.

[10] McCarey, F., M.Ó. Cinnéide, and N. Kushmerick, Recommending Library Methods: An Evaluation of the Vector Space Model (VSM) and Latent Semantic Indexing (LSI), in *Proceedings of 2006 International Conference on Software Reuse*. 2006. p. 217-230.

[11] McDonald, D.W. and M.S. Ackerman, Expertise Recommender: A Flexible Recommendation System Architecture, in *Proceedings of CSCW 2000*. 2000. p. 101-120.

[12] Mili, A., et al., Toward an Engineering Discipline of Software Reuse. *IEEE Software*, 1999. **16**(5): p. 22-31.

[13] Mockus, A. and J. Herbsleb, Expertise Browser: A Quantitative Approach to Identifying Expertise, in *Proceedings of 2002 International Conference on Software Engineering*. 2002: Orlando, FL. p. 503-512.

[14] Nahapiet, J. and S. Ghoshal, Social Capital, Intellectual Capital, and the Organizational Advantage. *Academy of Management Review*, 1998. **23**: p. 242-266.

[15] Raymond, E.S., *The Art of UNIX Programming*. 2004, Boston, MA: Addison-Wesley.

[16] Robertson, S.E., et al., Okapi at TREC-3, in *The 3rd Text REtrieval Conference (TREC-3)*, D.K. Harman, Editor. 1995, National Institute of Standards and Technology: Gaithersburg, MD. p. 109-126.

[17] Sahavechaphan, N. and K. Claypool, XSnippet: Mining for Sample Code, in *Proceedings of the 21st ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*. 2006: Portland, OR. p. 413-430.

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

[19] van den Hoff, B., J. de Ridder, and E. Aukema, Exploring the Eagerness to Share Knowledge: The Role of Social Capital and ICT in Knowledge Sharing, in *Social Capital and Information Technology*, M. Huysman and V. Wulf, Editors. 2004, Cambridge, MA: The MIT Press. p. 163-186.

[20] Williams, M.D., What Makes RABBIT Run? in *International Journal of Man-Machine Studies*. 1984. p. 333-352.

[21] Ye, Y. and G. Fischer, Supporting Reuse by Delivering Task-Relevant and Personalized Information, in *Proceedings of 2002 International Conference on Software Engineering (ICSE'02)*. 2002: Orlando, FL. p. 513-523.

[22] Ye, Y. and G. Fischer, Reuse-Conducive Development Environments. *Automated Software Engineering*, 2005. **12**(2): p. 199-235.

[23] Ye, Y., Y. Yamamoto, and K. Nakakoji, A Socio-Technical Framework for Supporting Programmers, in *Proceedings of 2007 ACM Symposium on Foundations of Software Engineering (FSE2007)*. 2007 (forthcoming).